Table of Contents

- 1 INTRODUCTION
  - 1.1 Overview
  - 1.2 Feature Description
  - 1.3 Problem Statement
  - 1.4 Motivation
  - 1.5 Data Set Overview
    - 1.5.1 Shape of the Data
    - 1.5.2 First Few Observations
    - 1.5.3 Feature Data Types
    - 1.5.4 Feature Statistics Summary
    - 1.5.5 Summary
    - 1.5.6 Questions for EDA
- 2 EDA
  - 2.1 Data Distribution Visualization - Part 1
    - 2.1.1 Categorical Data Conversion
    - 2.1.2 Distribution vs. Weather, Season, Working Day
    - 2.1.3 Distribution vs. Temperature
  - 2.2 Missing Data Fields
  - 2.3 Feature Engineering - Part 1
  - 2.4 Data Distribution Visualization - Part 2
    - 2.4.1 Hourly Distribution
    - 2.4.2 Monthly Distribution
  - 2.5 Outliers Analysis
    - 2.5.1 Weather = 'Heavy Snow/Rain' outlier
    - 2.5.2 Zscore >4 Pruning
  - 2.6 Correlation Analysis
    - 2.6.1 Regression Plots vs. Temperature, Humidity and Windspeed
    - 2.6.2 Heatmap Plot
  - 2.7 Feature Engineering - Part 2
- 3 MODELLING
  - 3.1 Data and Function Definition
    - 3.1.1 Train/Validation/Test Split
    - 3.1.2 Function Definitions
  - 3.2 Linear Regression
  - 3.3 Regularization Model - Ridge
  - 3.4 Regularization Model - Lasso
  - 3.5 Ensemble Model - Random Forest
    - 3.5.1 Single Model for Working and Non-working days + Categorical Features

        - 3.5.2 Two Separate Models + Binary Vector Features via OneHotEncoder
        - 3.5.3 Two Separate Models for Working and Non-working days + Categorical Features
    - 3.6 Ensemble Method - Gradient Boost
        - 3.6.1 Two Separate Models + Binary Vector Features via OneHotEncoder
        - 3.6.2 Two Separate Models for Working and Non-working days + Categorical Features
    - 3.7 Ensemble Method - Adaboost
    - 3.8 Stacking using Linear Regression
        - 3.8.1 Preparing data for Stacking
    - 3.9 Stacking using Random Forest
    - 3.10 Stacking using Gradient Boost
- 4 SUMMARY AND CONCLUSIONS
    - 4.1 RMSLE
    - 4.2 Train/Test Time
    - 4.3 Kaggle Submission
    - 4.4 Summary
        - 4.4.1 Data Exploration Conclusions
        - 4.4.2 Modeling Conclusions
        - 4.4.3 Limitations and Scope for Future Work

INTRODUCTION

Overview

Kaggle competition: https://www.kaggle.com/c/bike-sharing-demand

Bike sharing systems are a means of renting bicycles where the process of obtaining membership, rental, and bike return is automated via a network of kiosk locations throughout a city. Using these systems, people are able rent a bike from a one location and return it to a different place on an as-needed basis. Currently, there are over 500 bike-sharing programs around the world.

Feature Description

The data set (https://www.kaggle.com/c/bike-sharing-demand/data) consists of two spreadsheets - 1. train.csv, containing data to train the prediction algorithm and 2. test.csv, containing data to test the prediction algorithm. The data fields in the train.csv are enumerated below

- datetime - hourly date + timestamp
- season - 1 = spring, 2 = summer, 3 = fall, 4 = winter
- holiday - whether the day is considered a holiday

- workingday - whether the day is neither a weekend nor holiday
- weather -
  - 1: Clear, Few clouds, Partly cloudy, Partly cloudy
  - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
  - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
  - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- temp - temperature in Celsius
- atemp - "feels like" temperature in Celsius
- humidity - relative humidity
- windspeed - wind speed
- casual - number of non-registered user rentals initiated
- registered - number of registered user rentals initiated
- count - number of total rentals

Problem Statement

The goal of this project is to combine the historical bike usage patterns with the weather data in order to forecast bike rental demand.

- Target Column to be predicted: 'count'
- Input Columns used as variables (8 columns): ['datetime', 'season', 'holiday', 'workingday', 'weather', 'temp', 'atemp', 'humidity', 'windspeed']
  - The other two columns (casual and registered) comprises of the split-up of the target column 'count'.

Motivation

Several bike/scooter ride sharing facilities (e.g., Bird, Capital Bikeshare, Citi Bike) have started up lately especially in metropolitan cities like San Francisco, New York, Chicago and Los Angeles, and one of the most important problem from a business point of view is to predict the bike demand on any particular day. While having excess bikes results in wastage of resource (both with respect to bike maintenance and the land/bike stand required for parking and security), having fewer bikes leads to revenue loss (ranging from a short term loss due to missing out on immediate customers to potential longer term loss due to loss in future customer base), Thus, having a estimate on the demands would enable efficient functioning of these companies.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import style
import seaborn as sns
import missingno as msno

# Ignore  the warnings
import warnings
```

```
warnings.filterwarnings('always')
warnings.filterwarnings('ignore')

#plt.style.use('fivethirtyeight')
sns.set_style("whitegrid")
sns.set_context("talk", font_scale=0.8)
```

Data Set Overview

Load the data

In [2]:
```
# Read the data
mydata = pd.read_csv('./Data/train.csv', parse_dates=True, index_col='datetime')
testdata = pd.read_csv('./Data/test.csv', parse_dates=True, index_col='datetime')
```

Shape of the Data

In [3]:
```
print('Shape of data: ', mydata.shape)
```

```
Shape of data:  (10886, 11)
```

The provided data consists of over 10k observations with 11 column variables (excluding the datetime column - which has been used as an index)

First Few Observations

In [4]:
```
mydata.head(3)
```

Out[4]:

datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registe
2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	
2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	
2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	

Feature Data Types

In [5]:
```
mydata.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 10886 entries, 2011-01-01 00:00:00 to 2012-12-19 23:00:00
Data columns (total 11 columns):
season        10886 non-null int64
holiday       10886 non-null int64
workingday    10886 non-null int64
weather       10886 non-null int64
temp          10886 non-null float64
atemp         10886 non-null float64
humidity      10886 non-null int64
windspeed     10886 non-null float64
casual        10886 non-null int64
registered    10886 non-null int64
count         10886 non-null int64
dtypes: float64(3), int64(8)
memory usage: 1020.6 KB
```

Data consists of 12 columns variables and all of them are Numeric Columns.

Feature Statistics Summary

Below table provides the statistical details for each column.

In [6]:
```
mydata.describe()
```

Out[6]:

	season	holiday	workingday	weather	temp	atemp	humid
count	10886.000000	10886.000000	10886.000000	10886.000000	10886.00000	10886.000000	10886.0000
mean	2.506614	0.028569	0.680875	1.418427	20.23086	23.655084	61.8864
std	1.116174	0.166599	0.466159	0.633839	7.79159	8.474601	19.2450
min	1.000000	0.000000	0.000000	1.000000	0.82000	0.760000	0.0000
25%	2.000000	0.000000	0.000000	1.000000	13.94000	16.665000	47.0000
50%	3.000000	0.000000	1.000000	1.000000	20.50000	24.240000	62.0000
75%	4.000000	0.000000	1.000000	2.000000	26.24000	31.060000	77.0000
max	4.000000	1.000000	1.000000	4.000000	41.00000	45.455000	100.0000

In [7]:
```
print(mydata.index[[0, -1]]) # Range of time stamp
```

```
DatetimeIndex(['2011-01-01 00:00:00', '2012-12-19 23:00:00'], dtype='datetime64[ns]',
name='datetime', freq=None)
```

In [8]:
```
print('Casual + Registered = Count? ', ~(mydata.casual + mydata.registered - mydata['
```

```
Casual + Registered = Count?  True
```

Summary

The below table summarizes the column content for the data

Column Name	Format	Range	Explanation
datetime	yyyy-mm-dd hh:mm:ss	2011-01-01 00:00:00 to 2012-12-19 23:00:00	hourly date + time stamp
season	int64	1 to 4	1 = Spring, 2 = Summer, 3 = Fall, 4 = Winter
holiday	int64	0 or 1	1 = Holiday, 0 = Not a Holiday
workingday	int64	0 or 1	1 = Neither a weekend nor holiday, 0 = Either a weekend or a holiday
weather	int64	1 to 4	1 = Clear, Few clouds, Partly cloudy, Partly cloudy 2 = Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist 3 = Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds 4 = Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
temp	float64	-0.82 to 41	Temperature in Celsius
atemp	float64	0.76 to 45.46	'Feels like" temperature in Celsius
humidity	int64	0 to 100	Relative humidity
windspeed	float64	0 to 57	Wind Speed
casual	int64	0 to 367	Number of non-registered user rentals initiated
registered	int64	0 to 886	Number of registered user rentals initiated
count	int64	1 to 977	Number of total rentals (casual + registered)

Questions for EDA

- The target column (count) which refers to the number of bikes that have been rented at that hour, ranges between 1 and 977 over the 2 year span.
- Mean(count) = 192, with median and 75% quantile = 145 and 284, respectively. This suggests that the 'count' distribution is more denser at lower values. This is expected as out of 24 hours, we would expect the bike demand/usage to be high for maximum of maybe 6 hours or so.
- Would expect the strongest correlation from hours component in the datetime column
- During weekdays most of the bikers would probably be the local commuters (more likely registered users) and during weekends and holidays, the majority of the bikers are more likely to be tourists (casual riders)

EDA

Data Distribution Visualization - Part 1

Categorical Data Conversion

From the above set of 8 variables, we notice that the 4 of those columns ['season', 'holiday', 'workingday', 'weather'] should be category data types. Converting these 4 features to categories

In [9]:
```
# Converting into categorical data
category_list = ['season', 'holiday', 'workingday', 'weather']
for var in category_list:
    mydata[var] = mydata[var].astype('category')
    testdata[var] = testdata[var].astype('category')
```

In [10]:
```
# Mapping numbers to understandable text
season_dict = {1:'Spring', 2:'Summer', 3:'Fall', 4:'Winter'}
weather_dict = {1:'Clear', 2:'Misty+Cloudy', 3:'Light Snow/Rain', 4:'Heavy Snow/Rain'}
mydata['season'] = mydata['season'].map(season_dict)
mydata['weather'] = mydata['weather'].map(weather_dict)

testdata['season'] = testdata['season'].map(season_dict)
testdata['weather'] = testdata['weather'].map(weather_dict)

mydata.head(n=3)
```

Out[10]:

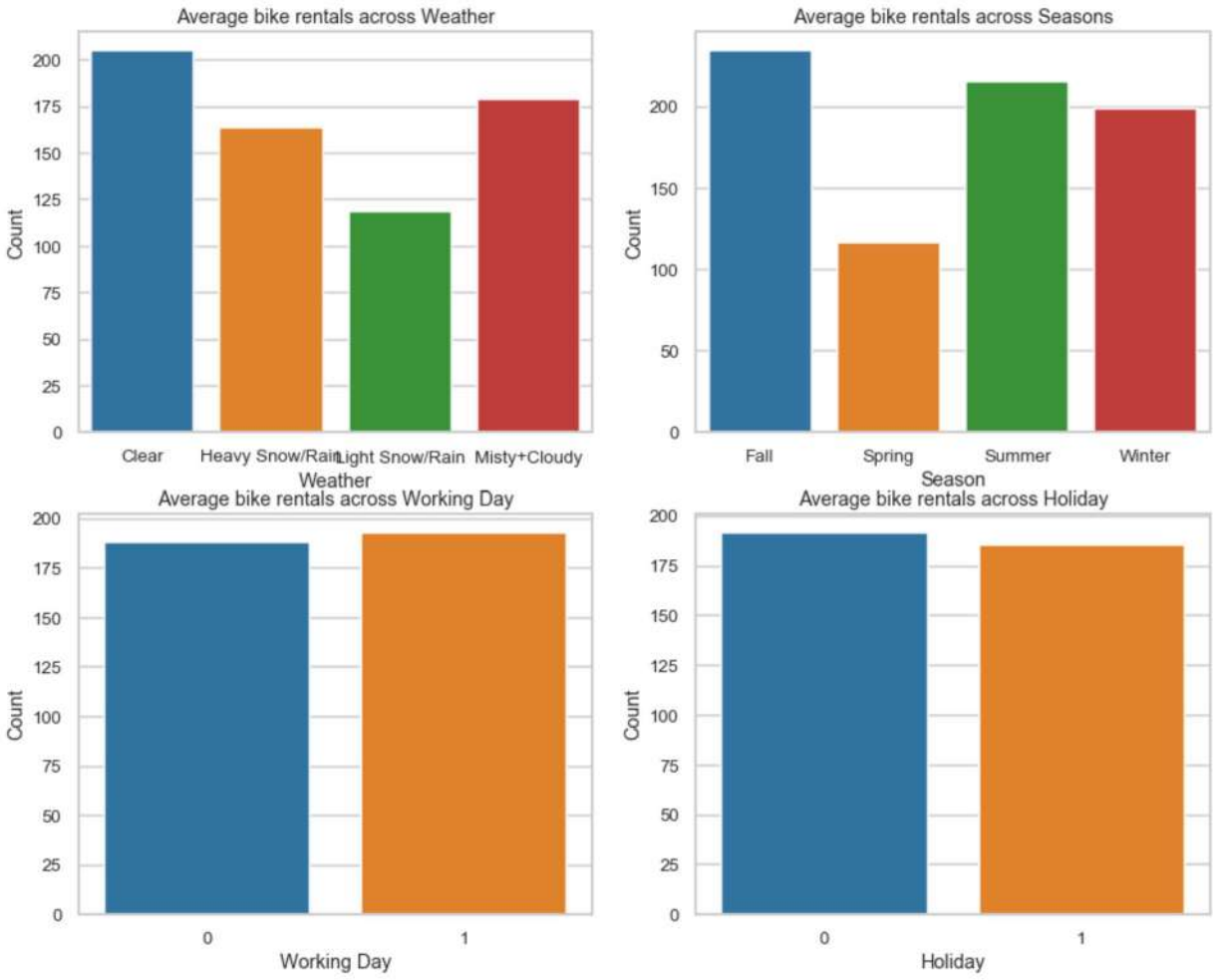
datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registe
2011-01-01 00:00:00	Spring	0	0	Clear	9.84	14.395	81	0.0	3	
2011-01-01 01:00:00	Spring	0	0	Clear	9.02	13.635	80	0.0	8	
2011-01-01 02:00:00	Spring	0	0	Clear	9.02	13.635	80	0.0	5	

Distribution vs. Weather, Season, Working Day

Let us see how 'count' = number of bikes rented varies across the various categorical data (weather, season, working day)

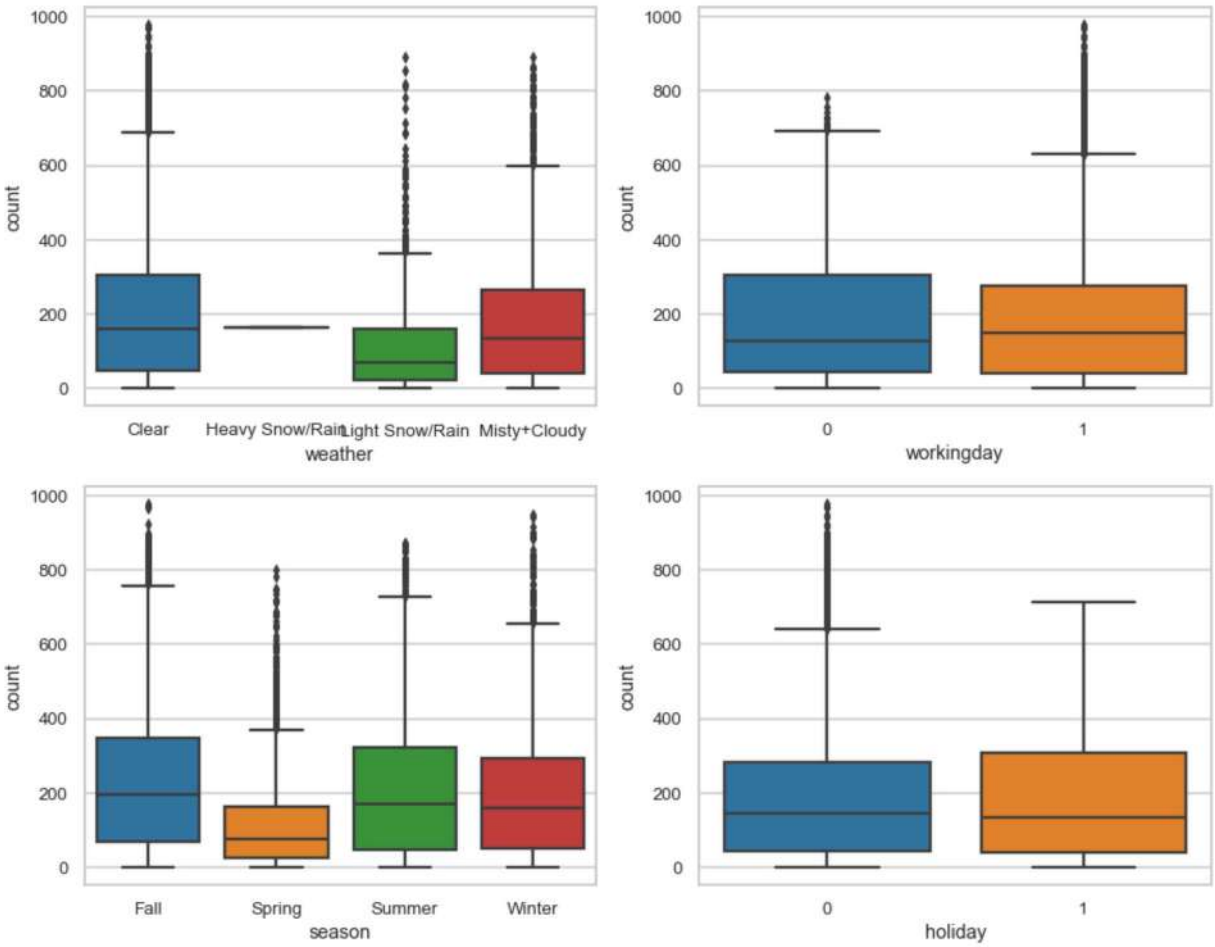
In [11]:
```
# Average values across each of the categorical columns
fig = plt.figure(figsize=(15, 12))
axes = fig.add_subplot(2, 2, 1)
group_weather = pd.DataFrame(mydata.groupby(['weather'])['count'].mean()).reset_index(
sns.barplot(data=group_weather, x='weather', y='count', ax=axes)
axes.set(xlabel='Weather', ylabel='Count', title='Average bike rentals across Weather'

axes = fig.add_subplot(2, 2, 2)
group_season = pd.DataFrame(mydata.groupby(['season'])['count'].mean()).reset_index()
sns.barplot(data=group_season, x='season', y='count', ax=axes)
```

```
axes.set(xlabel='Season', ylabel='Count', title='Average bike rentals across Seasons')

axes = fig.add_subplot(2, 2, 3)
group_workingday = pd.DataFrame(mydata.groupby(['workingday'])['count'].mean()).reset_
sns.barplot(data=group_workingday, x='workingday', y='count', ax=axes)
axes.set(xlabel='Working Day', ylabel='Count', title='Average bike rentals across Work

axes = fig.add_subplot(2, 2, 4)
group_season = pd.DataFrame(mydata.groupby(['holiday'])['count'].mean()).reset_index()
sns.barplot(data=group_season, x='holiday', y='count', ax=axes)
axes.set(xlabel='Holiday', ylabel='Count', title='Average bike rentals across Holiday'
plt.show()
```



Using seaborn boxplots to get an idea of the distribution and outliers acorss various categorical features

In [12]:
```
# Seaborn boxplots to get an idea of the distribution/outliers
f, axes = plt.subplots(2, 2, figsize=(15, 12))
hue_order= ['Clear', 'Heavy Snow/Rain', 'Light Snow/Rain', 'Misty+Cloudy']
sns.boxplot(data=mydata, y='count', x='weather', ax=axes[0][0], order=hue_order)
sns.boxplot(data=mydata, y='count', x='workingday', ax=axes[0][1])
hue_order= ['Fall', 'Spring', 'Summer', 'Winter']
sns.boxplot(data=mydata, y='count', x='season', ax=axes[1][0], order=hue_order)
sns.boxplot(data=mydata, y='count', x='holiday', ax=axes[1][1])

plt.show()
```



Few Observations

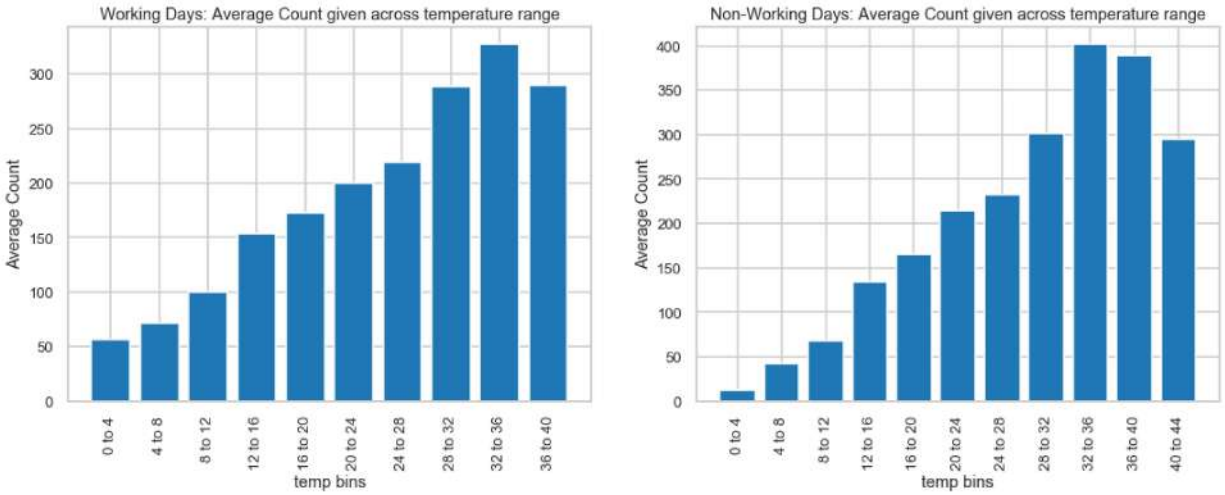
- Higher biker rentals as weather is more clear and sunny.
- Just '1 hour' instance where there were rentals under heavy rain/snow condition. Two possibilities
  - Could be an outlier
  - Reservations made at a time when the weather was good. But weather conditions logged sometime later in the same hour when the conditions were heavy rains/snow
- Bike reservations are lesser in Spring season compared to Summer and Fall
- Lots of outlier points for a particular seasons or weather conditions. This is most likely due to variable distribution across the day

Distribution vs. Temperature

Now let us see how number of bikes rented depends on the temperature

In [13]:
```
# Splitting data into working-day and non-working day
mydata_w = mydata[mydata.workingday==1]
mydata_nw = mydata[mydata.workingday==0]

bin_size = 4
mydata_w['temp_round'] = mydata_w['temp']//bin_size
mydata_nw['temp_round'] = mydata_nw['temp']//bin_size
```

```
mean_count_vs_temp_w = mydata_w.groupby('temp_round')['count'].mean()
mean_count_vs_temp_nw = mydata_nw.groupby('temp_round')['count'].mean()
idx_w, idx_nw = range(len(mean_count_vs_temp_w)), range(len(mean_count_vs_temp_nw))
labels_w = [str(bin_size*i)+' to '+str(bin_size*(i+1)) for i in range(len(mean_count_v
labels_nw = [str(bin_size*i)+' to '+str(bin_size*(i+1)) for i in range(len(mean_count_

fig = plt.figure(figsize=(18, 6))
axes = fig.add_subplot(1, 2, 1)
plt.bar(x=idx_w, height=mean_count_vs_temp_w)
plt.xticks(idx_w, labels_w, rotation=90)
plt.xlabel('temp bins')
plt.ylabel('Average Count')
plt.title('Working Days: Average Count given across temperature range')

axes = fig.add_subplot(1, 2, 2)
plt.bar(x=idx_nw, height=mean_count_vs_temp_nw)
plt.xticks(idx_nw, labels_nw, rotation=90)
plt.xlabel('temp bins')
plt.ylabel('Average Count')
plt.title('Non-Working Days: Average Count given across temperature range')

plt.show()
```



Observation

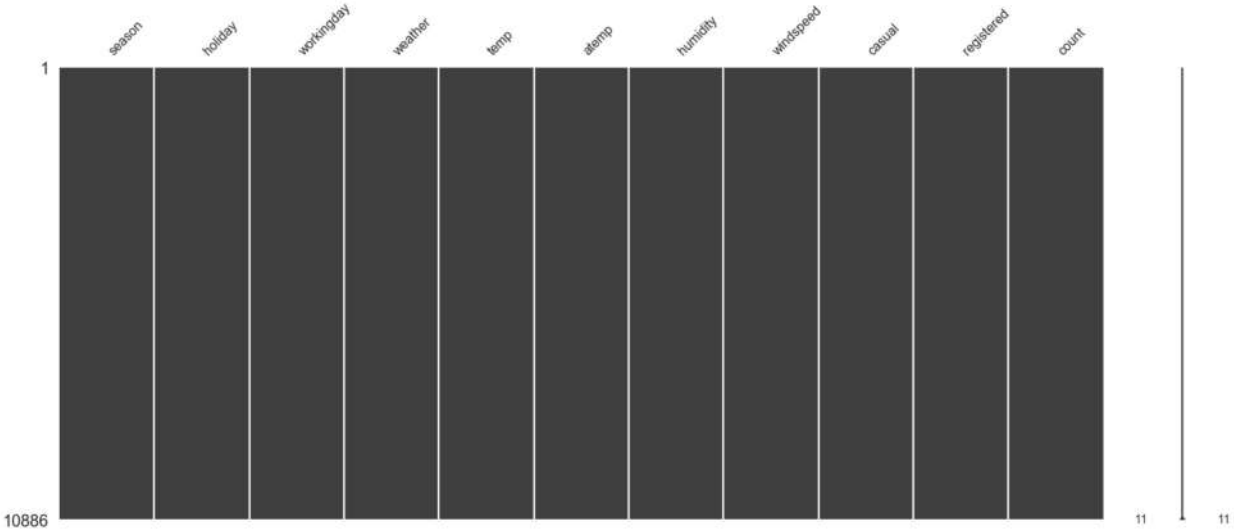
- From the above histogram plot, we can see that there is a steady increase in the average bikes rented with temperature with a small decrease at the highest temperature bin

Missing Data Fields

From the above .info() command, we notice that every column has 10886 (= number of rows) non-null values. This seems to be a very clean set of data and there are no missing data in any of the 'row x columns'.

In [14]:
```
msno.matrix(mydata)
```

Out[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2439a498550>
```



Feature Engineering - Part 1

Lets, split the datetime column into ['month', 'date', 'day', 'hour'] categories since the bike demand is more likely dependent on these individual categories. Creating these 4 additional category columns

In [15]:
```
# Splitting datetime object into month, date, hour and day category columns
mydata['month'] = [x.month for x in mydata.index]
mydata['date'] = [x.day for x in mydata.index]
mydata['hour'] = [x.hour for x in mydata.index]
mydata['day'] = [x.weekday() for x in mydata.index]

testdata['month'] = [x.month for x in testdata.index]
testdata['date'] = [x.day for x in testdata.index]
testdata['hour'] = [x.hour for x in testdata.index]
testdata['day'] = [x.weekday() for x in testdata.index]

category_list = ['month', 'date', 'hour', 'day']
for var in category_list:
    mydata[var] = mydata[var].astype('category')
    testdata[var] = testdata[var].astype('category')
```

Mapping 0 to 6 day indices to Monday to Saturday

In [16]:
```
# Mapping 0 to 6 day indices to Monday to Saturday
day_dict = {0:'Monday', 1:'Teusday', 2:'Wednesday', 3:'Thursday', 4:'Friday', 5:'Satur
mydata['day'] = mydata['day'].map(day_dict)
testdata['day'] = testdata['day'].map(day_dict)

mydata.head(n=3)
```

Out[16]:

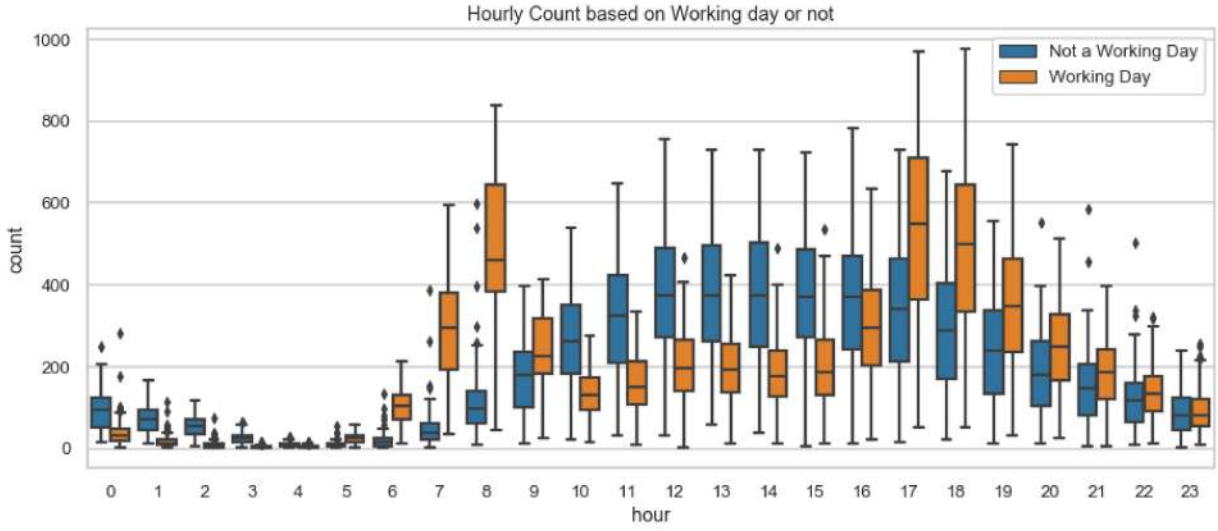
datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registe
2011-01-01 00:00:00	Spring	0	0	Clear	9.84	14.395	81	0.0	3	
2011-01-01 01:00:00	Spring	0	0	Clear	9.02	13.635	80	0.0	8	
2011-01-01 02:00:00	Spring	0	0	Clear	9.02	13.635	80	0.0	5	

Data Distribution Visualization - Part 2

Hourly Distribution

Now that we have separated out the hour data, let us plot the distribution across hour in a day. Clearly, we expect to see some pattern in the distribution - for example, people would tend to rent bikes early in the morning and return them in the evening.

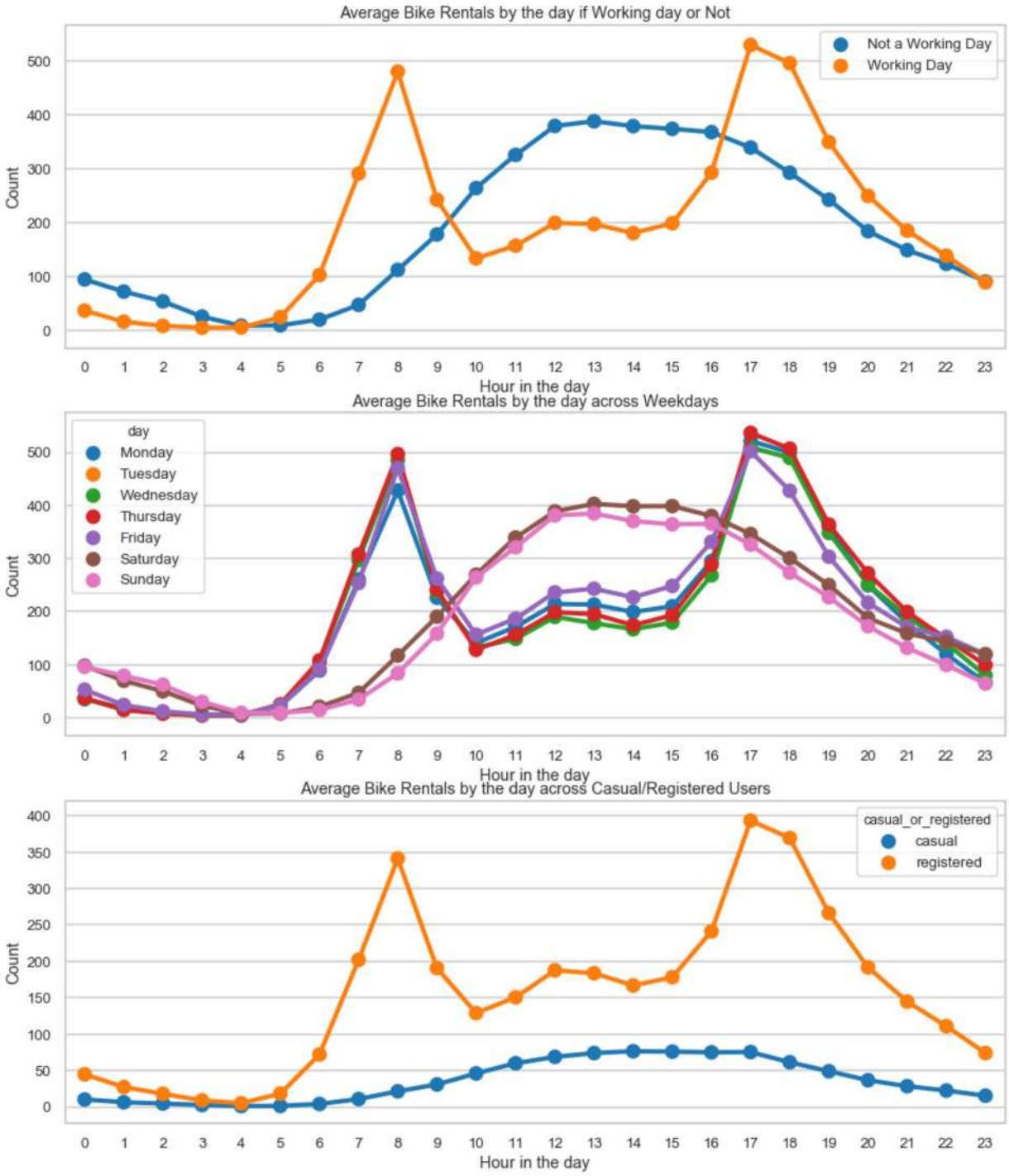
In [17]:
```
# seaborn boxplots across hours
f, axes = plt.subplots(1, 1, figsize=(15, 6))
sns.boxplot(data=mydata, y='count', x='hour', hue='workingday', ax=axes)
handles, _ = axes.get_legend_handles_labels()
axes.legend(handles, ['Not a Working Day', 'Working Day'])
axes.set(title='Hourly Count based on Working day or not')

plt.show()
```



Plotting average bike count for each hour as a function of various categories.

In [18]:
```
# Plots of average count across hour in a day for various categories
```

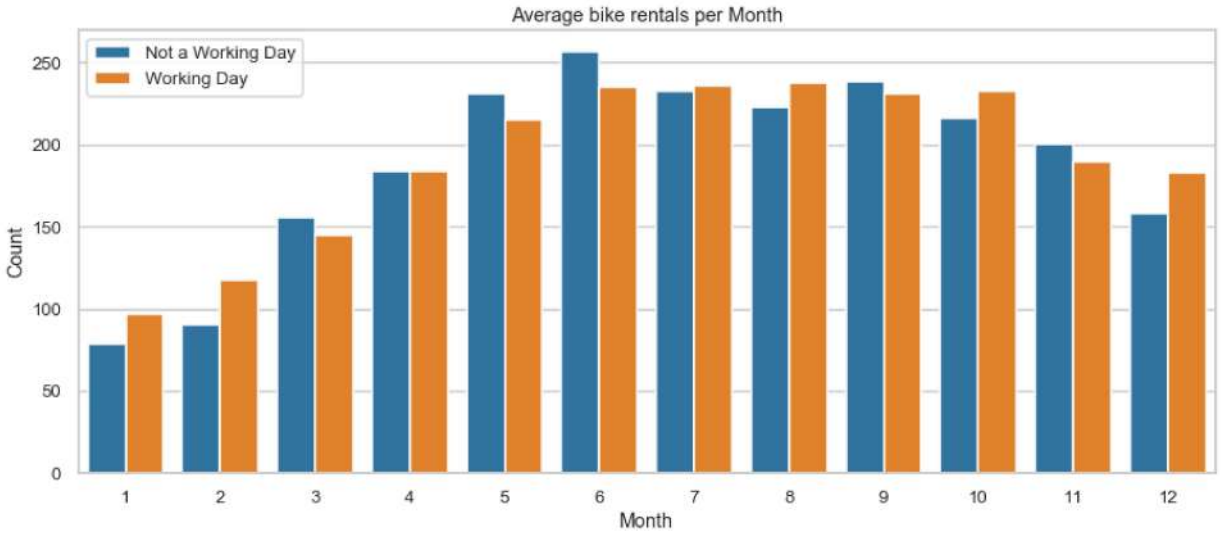
```
f, axes = plt.subplots(nrows=3, ncols=1, figsize=(15, 18))
group_work_hour = pd.DataFrame(mydata.groupby(['workingday', 'hour'])['count'].mean()
sns.pointplot(data=group_work_hour, x='hour', y='count', hue='workingday', ax=axes[0],
handles, _ = axes[0].get_legend_handles_labels()
axes[0].legend(handles, ['Not a Working Day', 'Working Day'])
axes[0].set(xlabel='Hour in the day', ylabel='Count', title='Average Bike Rentals by t

hue_order= ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunda
group_day_hour = pd.DataFrame(mydata.groupby(['day', 'hour'])['count'].mean()).reset_i
sns.pointplot(data=group_day_hour, x='hour', y='count', hue='day', ax=axes[1], hue_orc
axes[1].set(xlabel='Hour in the day', ylabel='Count', title='Average Bike Rentals by t

df_melt = pd.melt(frame=mydata, id_vars='hour', value_vars=['casual', 'registered'], v
group_casual_hour = pd.DataFrame(df_melt.groupby(['hour', 'casual_or_registered'])['co
sns.pointplot(data=group_casual_hour, x='hour', y='count', hue='casual_or_registered',
axes[2].set(xlabel='Hour in the day', ylabel='Count', title='Average Bike Rentals by t

plt.show()
```



Monthly Distribution

Now let us look at the average count of bike rentals across the months

In [19]:
```
# Average Monthly Count Distribution plot
f, axes = plt.subplots(nrows=1, ncols=1, figsize=(15, 6))
group_month = pd.DataFrame(mydata.groupby(['month', 'workingday'])['count'].mean()).re
sns.barplot(data=group_month, x='month', y='count', hue='workingday', ax=axes)
axes.set(xlabel='Month', ylabel='Count', title='Average bike rentals per Month')
handles, _ = axes.get_legend_handles_labels()
axes.legend(handles, ['Not a Working Day', 'Working Day'])
plt.show()
```

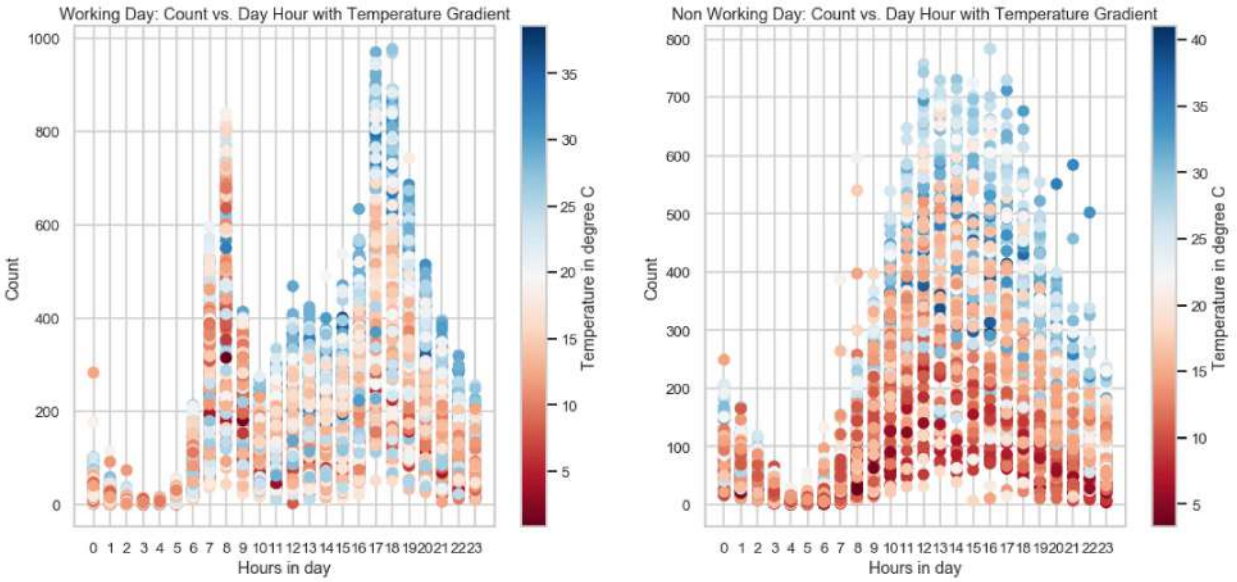


Observations

- Lesser number of outliers can be seen in the seaborn box plot across hours indicating that most of the outliers are due to highly varying distribution during the day
- Higher reservations can be seen at around 8am and 5pm (office hours) and close to 0 reservations very early in the morning
- From the above plot we can see the 2 patterns across the hours in a day in bike rentals
  - Working Day: First pattern where there is a peak in the rentals at around 8am and another at around 5pm. These correspond to working local bikers who typically are `registered` and go to work on `working day` which are `Monday` to `Friday`
  - Non Working Day: Second pattern where there is more or less a uniform rentals across the day with a peak at around noon time. These correspond to probably tourists who typically are `casual` users who rent/drop off bikes uniformly during the day and tour the city of Washington on `non working days` which typically are `Saturday` and `Sunday`
- Also, we can see that we have more bike rentals during the Fall (July to September) and Summer (April to June) Season.

In [20]:
```
mydata_w = mydata[mydata.workingday==1]
mydata_nw = mydata[mydata.workingday==0]

fig = plt.figure(figsize=(18, 8))
# Working Day
axes = fig.add_subplot(1, 2, 1)
f = axes.scatter(mydata_w.hour, mydata_w['count'], c=mydata_w.temp, cmap = 'RdBu')
axes.set(xticks = range(24), xlabel='Hours in day', ylabel='Count', title='Working Day
cbar = plt.colorbar(f)
cbar.set_label('Temperature in degree C')

# Non Working Day
axes = fig.add_subplot(1, 2, 2)
f = axes.scatter(mydata_nw.hour, mydata_nw['count'], c=mydata_nw.temp, cmap = 'RdBu')
axes.set(xticks = range(24), xlabel='Hours in day', ylabel='Count', title='Non Working
cbar = plt.colorbar(f)
cbar.set_label('Temperature in degree C')
```

```
plt.show()
```



Observations

- From the above, we can see that in general, more people tend to prefer biking at moderate to high temperatures; however, if the temperature is too hot (darkest of the blue dots), there is a small decline in count

Outliers Analysis

Weather = 'Heavy Snow/Rain' outlier

We had earlier seen a single observation with 'Heavy Snow/Rain' recording. Let us extract that observation to see if it is an outlier and how we should handle that data point.

In [21]:
```
heavy_weather_data = mydata.loc[mydata['weather']=='Heavy Snow/Rain', :]
print(heavy_weather_data.index)
mydata['2012-01-09 08:00' : '2012-01-09 20:00']
```

```
DatetimeIndex(['2012-01-09 18:00:00'], dtype='datetime64[ns]', name='datetime', freq=
None)
```

Out[21]:

datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	re
2012-01-09 08:00:00	Spring	0	1	Misty+Cloudy	9.02	12.880	55	6.0032	13	
2012-01-09 09:00:00	Spring	0	1	Misty+Cloudy	9.02	13.635	64	0.0000	9	
2012-01-09 10:00:00	Spring	0	1	Clear	9.84	14.395	60	0.0000	13	
2012-01-09 11:00:00	Spring	0	1	Misty+Cloudy	10.66	14.395	56	6.0032	6	
2012-01-09 12:00:00	Spring	0	1	Misty+Cloudy	10.66	13.635	56	7.0015	10	
2012-01-09 13:00:00	Spring	0	1	Misty+Cloudy	10.66	13.635	56	7.0015	3	
2012-01-09 14:00:00	Spring	0	1	Light Snow/Rain	9.02	11.365	75	11.0014	5	
2012-01-09 15:00:00	Spring	0	1	Light Snow/Rain	9.02	11.365	75	11.0014	5	
2012-01-09 16:00:00	Spring	0	1	Light Snow/Rain	9.02	12.880	87	6.0032	3	
2012-01-09 17:00:00	Spring	0	1	Light Snow/Rain	9.02	13.635	87	0.0000	5	
2012-01-09 18:00:00	Spring	0	1	Heavy Snow/Rain	8.20	11.365	86	6.0032	6	
2012-01-09 19:00:00	Spring	0	1	Light Snow/Rain	8.20	11.365	93	6.0032	3	
2012-01-09 20:00:00	Spring	0	1	Misty+Cloudy	8.20	11.365	86	6.0032	5	

From the above data, it appears like the weather was indeed bad at 6pm. It was bright and sunny at 10am and slowly got bad (weather = Misty->Light Snow->Heavy Snow) towards the end of the day. Also, there were several registered users who got to work that morning (407 commuters @ 8am). This explains why there were 158 registered commuters even under the worst of the weather conditions

Since there is just one occurence of Heavy Snow/Rain Condition, let us replace the Heavy Snow/Rain label to Light Snow/Rain.

In [22]:
```
# Replacing Heavy/Snow Rain condition with Light Snow/Rain
mydata.loc[mydata['weather']=='Heavy Snow/Rain', 'weather'] = 'Light Snow/Rain'
testdata.loc[testdata['weather']=='Heavy Snow/Rain', 'weather'] = 'Light Snow/Rain'

mydata['2012-01-09 18:00' : '2012-01-09 18:00']
```

Out[22]:

datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	regis
2012-01-09 18:00:00	Spring	0	1	Light Snow/Rain	8.2	11.365	86	6.0032	6	

Zscore >4 Pruning

Let us first take a look at the data entries with zscore > 4, i.e., data with more than 4 standard deviation away from the mean.

In [23]:
```
# Function to calculate zscore
def zscore(series):
    return (series-series.mean())/series.std()

mydata['count_zscore'] = mydata.groupby(['hour', 'workingday'])['count'].transform(zsc
outlier_idx = np.abs(mydata['count_zscore'])>4
outlier_data = mydata.loc[outlier_idx, :]
print('Shape of the outlier data entries: ', outlier_data.shape)
outlier_data
```

```
Shape of the outlier data entries:  (15, 16)
```

Out[23]:

datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual
2011-05-02 00:00:00	Summer	0	1	Clear	18.86	22.725	72	8.9981	68
2011-05-02 01:00:00	Summer	0	1	Clear	18.86	22.725	72	8.9981	41
2011-07-15 02:00:00	Fall	0	1	Clear	24.60	28.790	78	11.0014	16
2012-04-01 06:00:00	Summer	0	0	Misty+Cloudy	14.76	17.425	76	8.9981	9
2012-04-16 06:00:00	Summer	1	0	Clear	21.32	25.000	83	11.0014	10
2012-04-16 07:00:00	Summer	1	0	Clear	21.32	25.000	83	8.9981	20
2012-04-16 08:00:00	Summer	1	0	Misty+Cloudy	22.96	26.515	83	11.0014	48
2012-07-04 21:00:00	Fall	1	0	Clear	34.44	40.150	53	8.9981	222
2012-07-04 22:00:00	Fall	1	0	Clear	33.62	39.395	56	15.0013	175
2012-09-09 05:00:00	Fall	0	0	Clear	21.32	25.000	77	12.9980	16
2012-11-07 00:00:00	Winter	0	1	Misty+Cloudy	12.30	14.395	56	19.0012	49
2012-11-07 01:00:00	Winter	0	1	Misty+Cloudy	11.48	13.635	61	16.9979	6
2012-11-07 02:00:00	Winter	0	1	Misty+Cloudy	11.48	12.880	56	19.9995	6
2012-11-12 07:00:00	Winter	1	0	Clear	16.40	20.455	87	0.0000	16
2012-11-12 08:00:00	Winter	1	0	Clear	17.22	21.210	82	11.0014	50

All the outliers occur mostly early in the morning or late at night. Let us prune out these outliers. These could be due to some late night shows or holiday or some party.

In [24]:
```
# Removing outliers from mydata
mydata_without_outliers = mydata.loc[~outlier_idx, :]
print('Shape of data before outliner pruning: ', mydata.shape)
print('Shape of data after outlier pruning: ', mydata_without_outliers.shape)
```

```
Shape of data before outliner pruning:  (10886, 16)
Shape of data after outlier pruning:  (10871, 16)
```

Dropping the zscore column from the data frame

In [25]:
```
# Dropping the zscore column
mydata_without_outliers = mydata_without_outliers.drop('count_zscore', axis=1)
mydata_without_outliers.head(n=3)
```

Out[25]:

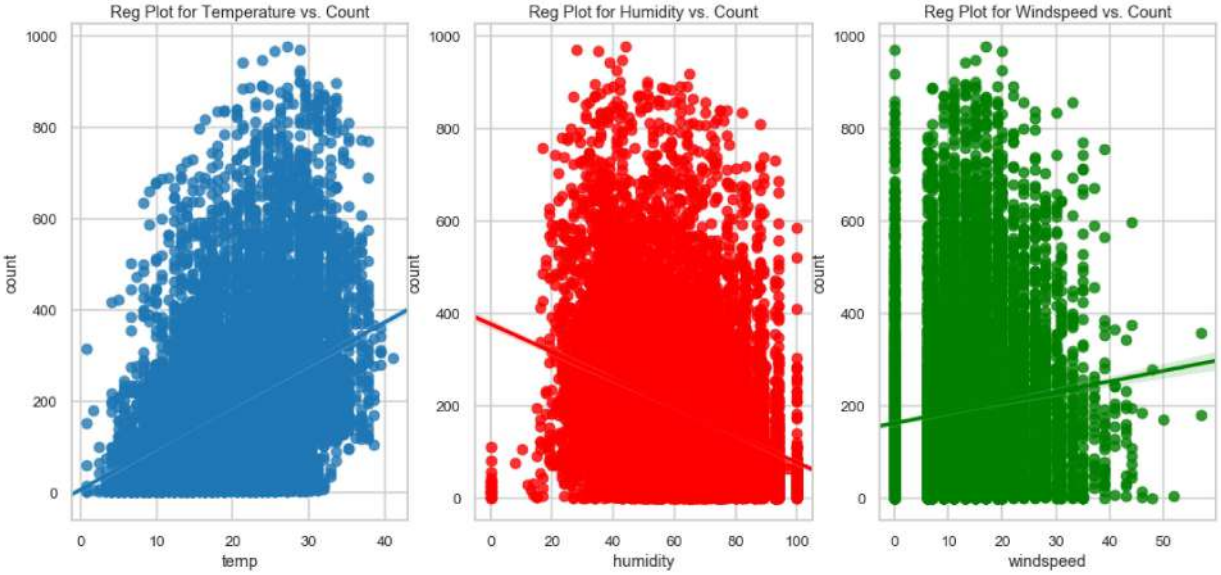
datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registe
2011-01-01 00:00:00	Spring	0	0	Clear	9.84	14.395	81	0.0	3	
2011-01-01 01:00:00	Spring	0	0	Clear	9.02	13.635	80	0.0	8	
2011-01-01 02:00:00	Spring	0	0	Clear	9.02	13.635	80	0.0	5	

Correlation Analysis

Regression Plots vs. Temperature, Humidity and Windspeed

Using seaborn to get regression plots with respect to Temperature, Humidity and Windspeed.

In [26]:
```
# Regression Plots with respect to Temperature, Humidity and Windspeed
fig = plt.figure(figsize=(18, 8))
axes = fig.add_subplot(1, 3, 1)
sns.regplot(data=mydata_without_outliers, x='temp', y='count',ax=axes)
axes.set(title='Reg Plot for Temperature vs. Count')
axes = fig.add_subplot(1, 3, 2)
sns.regplot(data=mydata_without_outliers, x='humidity', y='count',ax=axes, color='r')
axes.set(title='Reg Plot for Humidity vs. Count')
axes = fig.add_subplot(1, 3, 3)
sns.regplot(data=mydata_without_outliers, x='windspeed', y='count',ax=axes, color='g')
axes.set(title='Reg Plot for Windspeed vs. Count')
plt.show()
```



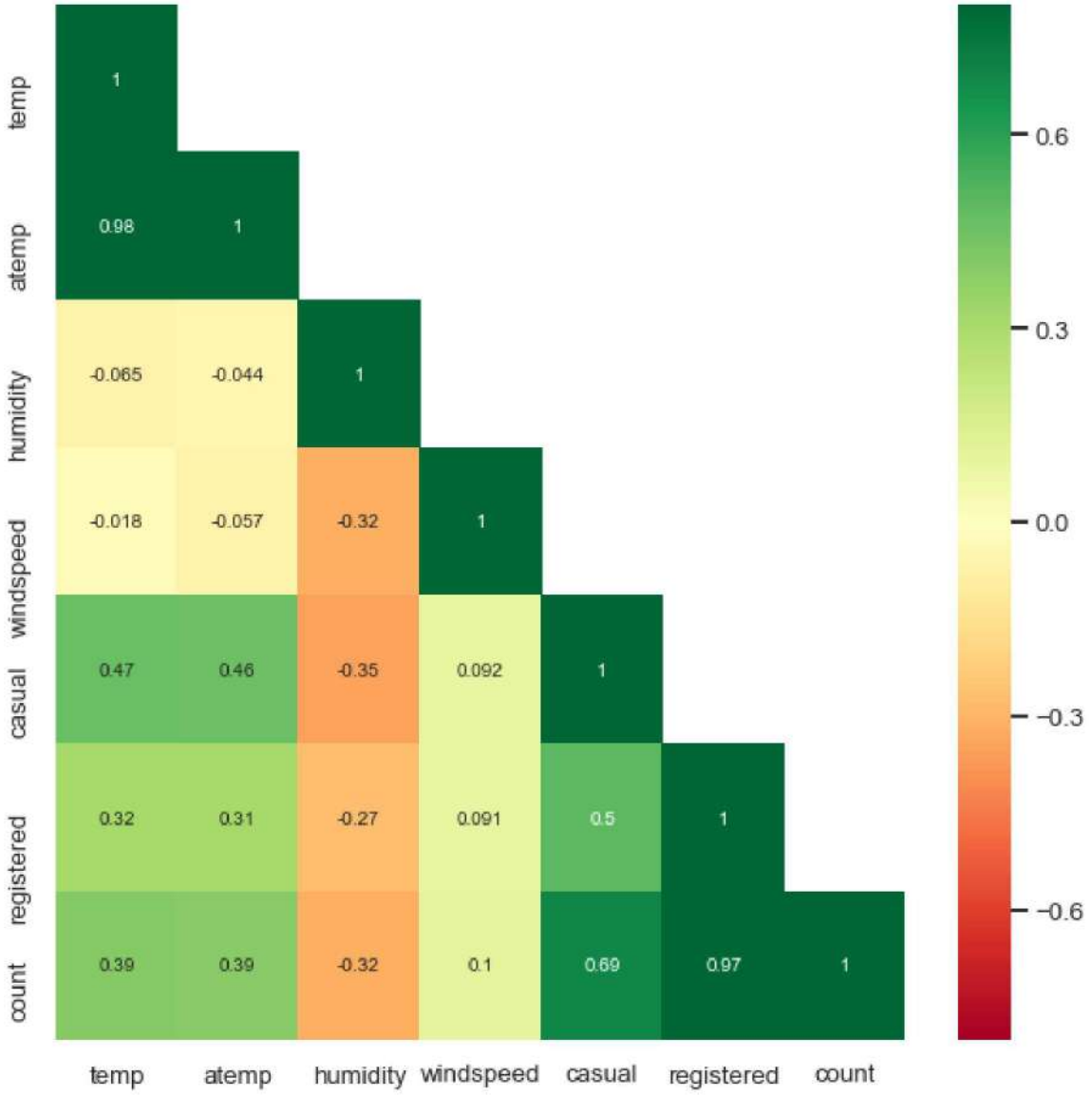
Observations

- The above regplot indicates a positive correlation of count with temperature and windspeed and a negative correlation with humidity

Heatmap Plot

Using heatmap plots of all the numerical features to obtain correlation of the bike rental with other numerical features

In [27]:
```
# Heatmap relative to all numeric columns
corr_matrix = mydata_without_outliers.corr()
mask = np.array(corr_matrix)
mask[np.tril_indices_from(mask)] = False

fig = plt.figure(figsize=(10, 10))
sns.heatmap(corr_matrix, mask=mask, annot=True, cbar=True, vmax=0.8, vmin=-0.8, cmap='
plt.show()
```



Inferences from the above heatmap

- temp (true temperature) and atemp (feels like temperature) are highly correlated as expected
- count is highly correlated with casual and registered as expected since count = casual + registered
- We see a positive correlation between count and temperature (as was seen in the regplot). This is probably only true for the range of temperatures provided
- We see a negative correlation between count and humidity. The more the humidity, the less people prefer to bike
- Not a great amount of correlation between humidity and temperature, though
- Count has a weak dependence on windspeed

Feature Engineering - Part 2

We have lots of categorical columns. We will transform each relevant and important categorical columns into binary vector columns. Later drop all the columns that are not required (redundant or very low correlation).

Summary of the column transformations

- Season: Month column has a direct mapping with season (Winter: January to March, Summer: April to June, Fall: July to September and Spring: October to December). Hence we will drop season column
- Holiday and 'day': workingday = weekday and not a holiday. Since we noticed that there were two kinds of bike rental behavoirs - during working days and not a working day, we will retain only the workingday column and drop 'day' and 'holiday' column
- Workingday: After observing the bike rental trend, we propose to build 2 separate models for 1. if it is a working day, and 2. if it is a non-working day. Hence, we can separate out the data based on this column and drop the column
- Weather: Split weather column to weather_1, weather_2 and weather 3 (recall that we had relabelled all the weather = 4 data points to weather = 3 due to its sparseness). Drop weather_3 since it is a function of the rest of the weather columns
- Temp: temp and atemp are highly correlated. Hence retain only the temp column
- Windspeed: Very poorly correlated with count. Hence drop this column
- Casual and registered: These are individual components of our to be predicted column (count). Hence drop these columns
- Month: Split month column to month_1, month_2, ..., month_12. Drop month_12 since it is a function of the rest of the month columns
- Date: Intuitively, there is should be no dependency on date. Hence drop this column
- Hour: Split hour column to hour_0, hour_1, ..., hour_23. Drop hour_23 since it is a function of the rest of the hour columns

In [28]:
```
# Using numbers to represent categorical data to transform the categorical columns
season_inv_dict = {'Spring':1, 'Summer':2, 'Fall':3, 'Winter':4}
weather_inv_dict = {'Clear':1, 'Misty+Cloudy':2, 'Light Snow/Rain':3, 'Heavy Snow/Rair
day_inv_dict = {'Monday':0, 'Teusday':1, 'Wednesday':2, 'Thursday':3, 'Friday':4, 'Sat

mydata_without_outliers['season'] = mydata_without_outliers['season'].map(season_inv_c
mydata_without_outliers['weather'] = mydata_without_outliers['weather'].map(weather_ir
mydata_without_outliers['day'] = mydata_without_outliers['day'].map(day_inv_dict)

testdata['season'] = testdata['season'].map(season_inv_dict)
testdata['weather'] = testdata['weather'].map(weather_inv_dict)
testdata['day'] = testdata['day'].map(day_inv_dict)

# Dropping columns from the provided data set that are either highly correlated with t
# season with month, holiday and day with workingday, temp with atemp
# or poorly correlated with the target column: windspeed and date
drop_columns_1 = ['season', 'holiday', 'atemp', 'windspeed', 'date', 'day']
mydata_without_outliers = mydata_without_outliers.drop(drop_columns_1, axis=1)
testdata = testdata.drop(drop_columns_1, axis=1)
mydata_without_outliers.head(n=3)
```

Out[28]:

datetime	workingday	weather	temp	humidity	casual	registered	count	month	hour
2011-01-01 00:00:00	0	1	9.84	81	3	13	16	1	0
2011-01-01 01:00:00	0	1	9.02	80	8	32	40	1	1
2011-01-01 02:00:00	0	1	9.02	80	5	27	32	1	2

Transforming all the categorical columns into binary columns...

In [29]:
```
# Transforming all the categorical columns into binary columns
month=pd.get_dummies(mydata_without_outliers['month'], prefix='month')
weather=pd.get_dummies(mydata_without_outliers['weather'], prefix='weather')
hour=pd.get_dummies(mydata_without_outliers['hour'], prefix='hour')
mydata_train=pd.concat([mydata_without_outliers, weather, month, hour],axis=1)

month=pd.get_dummies(testdata['month'], prefix='month')
weather=pd.get_dummies(testdata['weather'], prefix='weather')
hour=pd.get_dummies(testdata['hour'], prefix='hour')
mydata_test=pd.concat([testdata, weather, month, hour],axis=1)

mydata_train.columns
```

Out[29]:
```
Index(['workingday', 'weather', 'temp', 'humidity', 'casual', 'registered',
       'count', 'month', 'hour', 'weather_1', 'weather_2', 'weather_3',
       'month_1', 'month_2', 'month_3', 'month_4', 'month_5', 'month_6',
       'month_7', 'month_8', 'month_9', 'month_10', 'month_11', 'month_12',
       'hour_0', 'hour_1', 'hour_2', 'hour_3', 'hour_4', 'hour_5', 'hour_6',
       'hour_7', 'hour_8', 'hour_9', 'hour_10', 'hour_11', 'hour_12',
       'hour_13', 'hour_14', 'hour_15', 'hour_16', 'hour_17', 'hour_18',
       'hour_19', 'hour_20', 'hour_21', 'hour_22', 'hour_23'],
      dtype='object')
```

Dropping columns and the last binary vector column (which are fully correlated and can be expressed as a function of other columns)

In [30]:
```
# Dropping columns and the last binary vector column
drop_columns_2 = ['weather', 'month', 'hour', 'weather_3', 'month_12', 'hour_23']

mydata_train = mydata_train.drop(drop_columns_2+['casual', 'registered'], axis=1)
mydata_test = mydata_test.drop(drop_columns_2, axis=1)
mydata_without_outliers = mydata_without_outliers.drop(['casual', 'registered'], axis=

mydata_train.columns
```

Out[30]:
```
Index(['workingday', 'temp', 'humidity', 'count', 'weather_1', 'weather_2',
       'month_1', 'month_2', 'month_3', 'month_4', 'month_5', 'month_6',
       'month_7', 'month_8', 'month_9', 'month_10', 'month_11', 'hour_0',
       'hour_1', 'hour_2', 'hour_3', 'hour_4', 'hour_5', 'hour_6', 'hour_7',
       'hour_8', 'hour_9', 'hour_10', 'hour_11', 'hour_12', 'hour_13',
       'hour_14', 'hour_15', 'hour_16', 'hour_17', 'hour_18', 'hour_19',
       'hour_20', 'hour_21', 'hour_22'],
      dtype='object')
```

MODELLING

Since the bike rental trend is quite different between working and non-working day, we try the following two ways to solve this problem

1. Build 2 separate models - 1. for working day, and 2. for non-working day.
2. Build a single model using workingday as one of the features

Data and Function Definition

Train/Validation/Test Split

First split the labelled data provided into internal training and testing set. Kaggle has held out data from 20th to the end of the month (for every month) as test set. Following a similar approach, split the labelled data (mydata_train) into

1. Training set, model_train
   - This will contain data of from the 1st to 15th of every month
   - This will be used to train our model
2. Testing set, model_test
   - This will contain data from 16th to 19th of every month
   - This will be used to test our model

The final test data for which labels (count) have not been provided corresponds to 20th to the end of every month

In [31]:
```
model_train, model_test = mydata_train[mydata_train.index.day<15], mydata_train[mydata
model_train2, model_test2 = mydata_without_outliers[mydata_without_outliers.index.day<

# Separating out the working an non-working data from the training set
model_train_w = model_train[model_train['workingday']==1]
model_train_nw = model_train[model_train['workingday']==0]
model_train2_w = model_train2[model_train2['workingday']==1]
model_train2_nw = model_train2[model_train2['workingday']==0]

model_test_w = model_test[model_test['workingday']==1]
model_test_nw = model_test[model_test['workingday']==0]
model_test2_w = model_test2[model_test2['workingday']==1]
model_test2_nw = model_test2[model_test2['workingday']==0]

# Dropping workingday column
model_train_w = model_train_w.drop('workingday', axis=1)
model_train_nw = model_train_nw.drop('workingday', axis=1)
model_train2_w = model_train2_w.drop('workingday', axis=1)
model_train2_nw = model_train2_nw.drop('workingday', axis=1)

model_test_w = model_test_w.drop('workingday', axis=1)
model_test_nw = model_test_nw.drop('workingday', axis=1)
```

```
model_test2_w = model_test2_w.drop('workingday', axis=1)
model_test2_nw = model_test2_nw.drop('workingday', axis=1)
```

In [32]:
```
# Contains Binary Vector Form of features (Obtained from OneHotEncoder transformed cat
X, X_w, X_nw = model_train.drop('count', axis=1), model_train_w.drop('count', axis=1),
y, y_w, y_nw = model_train['count'], model_train_w['count'], model_train_nw['count']
logy, logy_w, logy_nw = np.log1p(y), np.log1p(y_w), np.log1p(y_nw)

Xtest, Xtest_w, Xtest_nw = model_test.drop('count', axis=1), model_test_w.drop('count'
ytest, ytest_w, ytest_nw = model_test['count'], model_test_w['count'], model_test_nw['
logytest, logytest_w, logytest_nw = np.log1p(y), np.log1p(y_w), np.log1p(y_nw)

# Contains Categorical features instead of the Binary Vector Form
X2, X2_w, X2_nw = model_train2.drop('count', axis=1), model_train2_w.drop('count', axi
y2, y2_w, y2_nw = model_train2['count'], model_train2_w['count'], model_train2_nw['cou
logy2, logy2_w, logy2_nw = np.log1p(y2), np.log1p(y2_w), np.log1p(y2_nw)

Xtest2, Xtest2_w, Xtest2_nw = model_test2.drop('count', axis=1), model_test2_w.drop('c
ytest2, ytest2_w, ytest2_nw = model_test2['count'], model_test2_w['count'], model_test
logytest2, logytest2_w, logytest2_nw = np.log1p(y), np.log1p(y_w), np.log1p(y_nw)

# Data Frame to store all the RMSLE scores for various algorithms
algo_score = pd.DataFrame()
algo_score.index.name = 'Modelling Algo'
algo_score['Train RMSLE (Working Day)'] = None
algo_score['Train RMSLE (Non Working Day)'] = None
algo_score['Train RMSLE (Average)'] = None
algo_score['Test RMSLE (Working Day)'] = None
algo_score['Test RMSLE (Non Working Day)'] = None
algo_score['Test RMSLE (Average)'] = None
algo_score['Validation RMSLE (Working Day)'] = None
algo_score['Validation RMSLE (Non Working Day)'] = None
algo_score['Validation RMSLE (Average)'] = None
algo_score['Hyperparameters-Working'] = None
algo_score['Hyperparameters-Non Working'] = None
algo_score['Training+Test Time (sec)'] = None
cv_time = []

# Data Frame for second level of prediction. Collect the predicted y values for traini
ypred_train = pd.DataFrame(index = X.index)
ypred_test = pd.DataFrame(index = Xtest.index)
ypred_train['count'], ypred_test['count'] = y, ytest
```

Function Definitions

Let us define few functions which we would call for every model

`rmsle` and `rmsle_log`: These are RMSLE (Root Mean Square Log Error) functions which will be used as our scoring function (also used by Kaggle)

In [33]:
```
from sklearn.metrics import make_scorer

# Metric used to measure the model (Root Mean Square Log Error)
def rmsle(y_actual, y_pred):
    log1 = np.nan_to_num(np.array([np.log1p(v) for v in y_pred]))
    log2 = np.nan_to_num(np.array([np.log1p(v) for v in y_actual]))
    calc = (log1 - log2) ** 2
```

```
    return np.sqrt(np.mean(calc))

# RMSLE function with inputs in log form. Used for CrossValidation scoring
def rmsle_log(logy_actual, logy_pred):
    calc = (logy_actual - logy_pred) ** 2
    return np.sqrt(np.mean(calc))
rmsle_cv = make_scorer(rmsle_log, greater_is_better=False)
```

plot_true_vs_pred : This is used to plot True and Predicted count values for a particular time interval

In [34]:
```
# Plots True vs. Predictied count values in a particular time interval
def plot_true_vs_pred (y_w_actual, y_nw_actual, y_w_pred, y_nw_pred, algo, t_from, t_t
    fig = plt.figure(figsize=(18, 16))

    # Working day plot
    axes = fig.add_subplot(2, 1, 1)
    axes.plot(y_w_actual[t_from:t_to], label='Actual', marker='.', markersize=15)
    axes.plot(y_w_pred[t_from:t_to], label='Predicted', marker='.', markersize=15)
    axes.set(xlabel='Time', ylabel='Count', title='{0} Model for Working Day: Count be
    axes.legend()

    # Non working day plot
    axes = fig.add_subplot(2, 1, 2)
    axes.plot(y_nw_actual[t_from:t_to], label='Actual', marker='.', markersize=15)
    axes.plot(y_nw_pred[t_from:t_to], label='Predicted', marker='.', markersize=15)
    axes.set(xlabel='Time', ylabel='Count', title='{0} Model for Non Working Day: Cour
    axes.legend()
    plt.show()
```

model_fit : Since we will be trying out various models to figure out which works best, let us customize a fit and predict function which will fit the trainging data and predict on the test data and return the required metrics.
The below function can fit/predict training and testing data for one of these two cases

1. working and non-working days or
2. combined working and non-working days.

The output returned by the function are

1. rmsle for training, test data for working, non-working and combined
2. predicted y for training, test for working/non-working or combined

In [35]:
```
# Customized Function to fit/predict training and testing data for:
# 1. working and non-working days or 2. combined working and non-working days
# Output: 1. rmsle for training, test data for working, non-working and combined
#         2. predicted y for training, test for working/non-working or combined
def model_fit (model_w, X_tr_w, X_t_w, y_tr_w, y_t_w, model_nw=None, X_tr_nw=None, X_t
    ''' Case 1: If separate models for Working day and non-working day
    model_w, model_nw = Models for Working and non-working days, respectively
    X_tr_w, y_tr_w = Training data set for Working days
    X_t_w, y_t_w = Testing data set for Working days
    X_tr_nw, y_tr_nw = Training data set for Non Working days
    X_t_nw, y_t_nw = Testing data set for Non Working days
```

```
    Case 2: If single model for working day and non-working day
    model_w = Single Model for Working and non-working days
    X_tr_w, y_tr_w = Training data set containing both Working and non-working days (f
    X_t_w, y_t_w = Testing data set containing both Working and non-working days'''

    # Working Day Modeling of Single Model for Working and Non-Working Day
    model_w.fit(X_tr_w, np.log1p(y_tr_w))
    logy_tr_w_predict = model_w.predict(X_tr_w)
    logy_t_w_predict = model_w.predict(X_t_w)

    y_tr_w_predict = np.expm1(logy_tr_w_predict)
    y_t_w_predict = np.expm1(logy_t_w_predict)

    rmsle_w_tr = rmsle(y_tr_w, y_tr_w_predict)
    rmsle_w_t = rmsle(y_t_w, y_t_w_predict)

    if model_nw is None:
        # Single Model for working and non-working days. The feature list in X should
        [rmsle_avg_tr, rmsle_avg_t] = [rmsle_w_tr, rmsle_w_t] # The RMSLE computed by
        rmsle_w_tr = rmsle(y_tr_w[X_tr_w.workingday==1], y_tr_w_predict[X_tr_w.working
        rmsle_nw_tr = rmsle(y_tr_w[X_tr_w.workingday==0], y_tr_w_predict[X_tr_w.workir
        rmsle_w_t = rmsle(y_t_w[X_t_w.workingday==1], y_t_w_predict[X_t_w.workingday==
        rmsle_nw_t = rmsle(y_t_w[X_t_w.workingday==0], y_t_w_predict[X_t_w.workingday=
        y_tr_nw_predict, y_t_nw_predict = None, None
    else:
        # Non-working day Modeling
        model_nw.fit(X_tr_nw, np.log1p(y_tr_nw))
        logy_tr_nw_predict = model_nw.predict(X_tr_nw)
        logy_t_nw_predict = model_nw.predict(X_t_nw)

        y_tr_nw_predict = np.expm1(logy_tr_nw_predict)
        y_t_nw_predict = np.expm1(logy_t_nw_predict)

        rmsle_nw_tr = rmsle(y_tr_nw, y_tr_nw_predict)
        rmsle_nw_t = rmsle(y_t_nw, y_t_nw_predict)

        # Combined RMSLE
        [rmsle_avg_tr, rmsle_avg_t] = [rmsle(np.concatenate([y_tr_w, y_tr_nw]), np.cor
                                       rmsle(np.concatenate([y_t_w, y_t_nw]), np.conca

    rmsle_all = [rmsle_w_tr, rmsle_nw_tr, rmsle_avg_tr, rmsle_w_t, rmsle_nw_t, rmsle_a
    y_pred_all = [y_tr_w_predict, y_t_w_predict, y_tr_nw_predict, y_t_nw_predict]

    return(rmsle_all, y_pred_all)
```

cros_val : This function is used to split the data into cv folds, train on cv-1 folds, test on the left out fold and most importantly save the predicted values of the left out fold. THese predictions are used as inputs for stacking.

The 5 folds used are - (1, 3), (4, 6), (7, 9), (10, 12) and (13, 15) of every month

In [36]:
```
def cross_val(model_w, X_in_w, y_in_w, model_nw=None, X_in_nw=None, y_in_nw=None, cv=5
    y_val_pred_w = pd.Series(index=y_in_w.index)
    y_val_pred_nw = None if model_nw == None else pd.Series(index=y_in_nw.index)
    for idx in range(cv):
        from_, to_ = idx*15/cv, (idx+1)*15/cv

        val_idx_w = (X_in_w.index.day>from_) & (X_in_w.index.day<=to_)
```

```
        train_idx_w = ~val_idx_w

        X_idx_w, y_idx_w, X_val_idx_w = X_in_w[train_idx_w], y_in_w[train_idx_w], X_ir
        model_w.fit(X_idx_w, np.log1p(y_idx_w))
        logy_val_pred_idx_w = model_w.predict(X_val_idx_w)
        y_val_pred_w[val_idx_w] = np.expm1(logy_val_pred_idx_w)

        if model_nw is not None:
            val_idx_nw = (X_in_nw.index.day>from_) & (X_in_nw.index.day<=to_)
            train_idx_nw = ~val_idx_nw

            X_idx_nw, y_idx_nw, X_val_idx_nw = X_in_nw[train_idx_nw], y_in_nw[train_ic
            model_nw.fit(X_idx_nw, np.log1p(y_idx_nw))
            logy_val_pred_idx_nw = model_nw.predict(X_val_idx_nw)
            y_val_pred_nw[val_idx_nw] = np.expm1(logy_val_pred_idx_nw)

    if model_nw is None:
        rmsle_avg = rmsle(y_in_w, y_val_pred_w)
        rmsle_w = rmsle(y_in_w[X_in_w.workingday==1], y_val_pred_w[X_in_w.workingday==
        rmsle_nw = rmsle(y_in_w[X_in_w.workingday==0], y_val_pred_w[X_in_w.workingday=
    else:
        rmsle_w = rmsle(y_in_w, y_val_pred_w)
        rmsle_nw = rmsle(y_in_nw, y_val_pred_nw)
        rmsle_avg = rmsle(np.concatenate([y_in_w, y_in_nw]), np.concatenate([y_val_pre

    rmsle_all = [rmsle_w, rmsle_nw, rmsle_avg]
    y_pred_all =[y_val_pred_w, y_val_pred_nw]
    return(rmsle_all, y_pred_all)
```

`stack_model_fit` : This function is used to fit and predict data sets for stacking models. These use the predicted values from individual models as inputs/features to train a new model

In [37]:
```
# Linear Regressor Ensemble for the above 3 models
def stack_model_fit (model, X_tr, X_t, y_tr, y_t):
    model.fit(X_tr, y_tr)
    y_tr_pred = model.predict(X_tr)
    y_t_pred = model.predict(X_t)

    [rmsle_avg_tr, rmsle_avg_t] = rmsle(y_tr, y_tr_pred), rmsle(y_t, y_t_pred)

    y_tr_w_pred, y_tr_nw_pred = y_tr_pred[X.workingday==1], y_tr_pred[X.workingday==0]
    y_t_w_pred, y_t_nw_pred = y_t_pred[Xtest.workingday==1], y_t_pred[Xtest.workingday
    y_tr_w, y_tr_nw = y_tr[X.workingday==1], y_tr[X.workingday==0]
    y_t_w, y_t_nw = y_t[Xtest.workingday==1], y_t[Xtest.workingday==0]

    rmsle_w_tr, rmsle_nw_tr = rmsle(y_tr_w, y_tr_w_pred), rmsle(y_tr_nw, y_tr_nw_pred)
    rmsle_w_t, rmsle_nw_t = rmsle(y_t_w, y_t_w_pred), rmsle(y_t_nw, y_t_nw_pred)

    rmsle_all = [rmsle_w_tr, rmsle_nw_tr, rmsle_avg_tr, rmsle_w_t, rmsle_nw_t, rmsle_a
    y_pred_all = [y_tr_pred, y_t_pred]

    return(rmsle_all, y_pred_all)
```

Linear Regression

Model Fit+Predict

In [38]:
```
from sklearn.linear_model import LinearRegression
lreg_w, lreg_nw = LinearRegression(), LinearRegression()

param_summary = ['', '', '']

rmsle_summary, y_predict_summary = model_fit(lreg_w, X_w, Xtest_w, y_w, ytest_w, lreg_
ypred_test.loc[Xtest.workingday==1,'LR'], ypred_test.loc[Xtest.workingday==0,'LR'] = y
```

In [39]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(lreg_w, X_w, y_w, lreg_nw, X_nw,
ypred_train.loc[X.workingday==1,'LR'], ypred_train.loc[X.workingday==0,'LR'] = y_predi

algo_score.loc['Linear Regression'] = rmsle_summary+rmsle_val_summary+param_summary
algo_score.loc[['Linear Regression']]
```

Out[39]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Val (A
Linear Regression	0.418155	0.432817	0.422797	0.390881	0.495181	0.428666	0.430658	0.474808	0

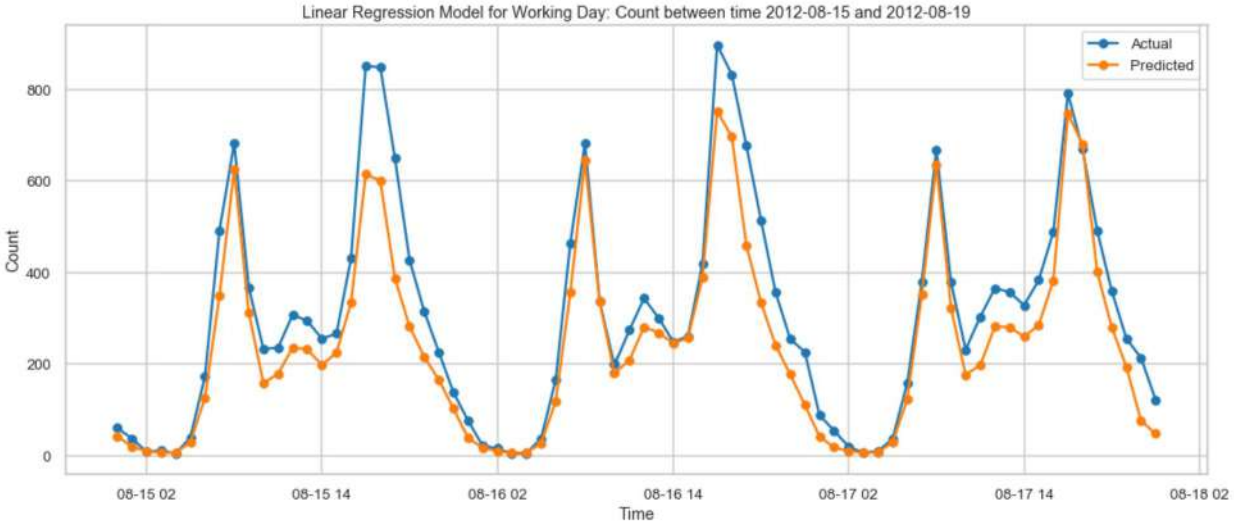
In [40]:
```
algo_score.loc['Linear Regression', 'Training+Test Time (sec)'] = 0.197
cv_time.append(0.237)
```

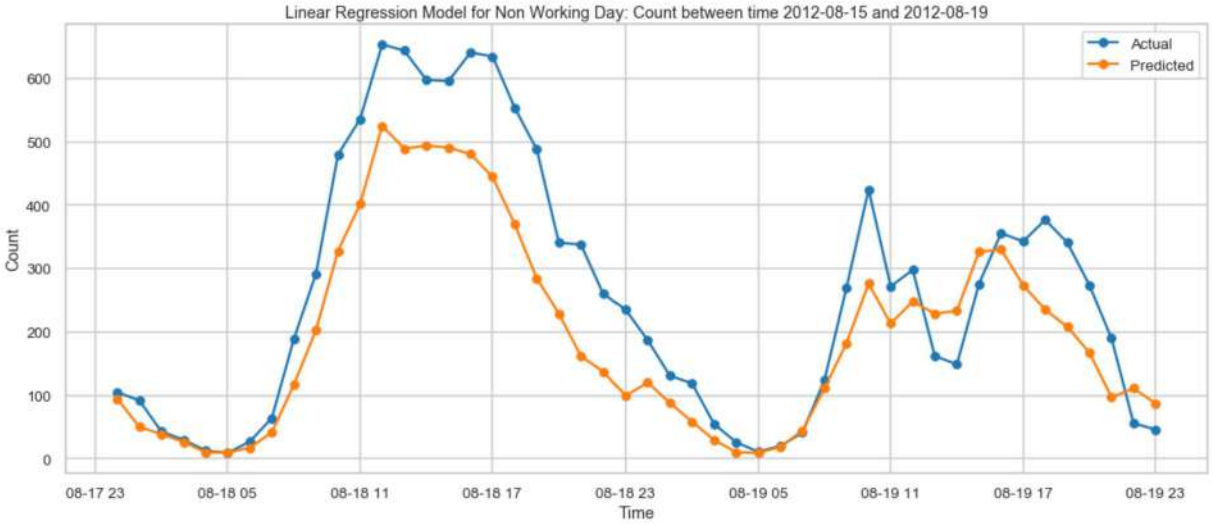
Observations

- Not an overfit model - both train and test RMSLE are more or less similar
- A good initial model

Prediction Plot for Test data in a month (last 5 days of a month)

In [41]:
```
# Linear Regression Plot: True vs. Predicted for one week
t_from, t_to = '2012-08-15', '2012-08-19'
ytest_w_predict, ytest_nw_predict = y_predict_summary[1], y_predict_summary[3]
ytest_w_predict = pd.Series(ytest_w_predict, index = ytest_w.index)
ytest_nw_predict = pd.Series(ytest_nw_predict, index = ytest_nw.index)

plot_true_vs_pred(ytest_w, ytest_nw, ytest_w_predict, ytest_nw_predict, 'Linear Regres
```





Storing the regression coefficents for comparison later...

In [42]:
```
# Features and the Estimated Linear Regression Coefficients obtained for Working day d
df_coeff = pd.DataFrame({'features': X_w.columns, 'Lin_Coeff_Working': lreg_w.coef_, '
```

Regularization Model - Ridge

Hyperparameter Tuning

In [43]:
```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

# Hyperparameter Tuning
param_grid = {'alpha': [0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100]}
ridge_w = GridSearchCV(Ridge(random_state=42), param_grid, cv=5, scoring=rmsle_cv)
ridge_w.fit(X_w, logy_w)
print('Best alpha for Working Day Ridge Regression Model: {}'.format(ridge_w.best_para
ridge_nw = GridSearchCV(Ridge(random_state=42), param_grid, cv=5, scoring=rmsle_cv)
ridge_nw.fit(X_nw, logy_nw)
print('Best alpha for Non Working Day Ridge Regression Model: {}'.format(ridge_nw.best
```

```
Best alpha for Working Day Ridge Regression Model: {'alpha': 10}
Best alpha for Non Working Day Ridge Regression Model: {'alpha': 10}
```

Model Fit + Predict

In [44]:
```
param_summary = [ridge_w.best_params_, ridge_nw.best_params_,'']

rmsle_summary, y_predict_summary = model_fit(ridge_w, X_w, Xtest_w, y_w, ytest_w, ridg
ypred_test.loc[Xtest.workingday==1,'Ridge'], ypred_test.loc[Xtest.workingday==0,'Ridge
```

In [45]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(ridge_w, X_w, y_w, ridge_nw, X_nw
ypred_train.loc[X.workingday==1,'Ridge'], ypred_train.loc[X.workingday==0,'Ridge'] = y

algo_score.loc['Ridge Regression'] = rmsle_summary+rmsle_val_summary+param_summary
algo_score.loc[['Ridge Regression']]
```

Out[45]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Val (A
Ridge Regression	0.423189	0.448141	0.431152	0.39423	0.516847	0.439148	0.438216	0.492361	

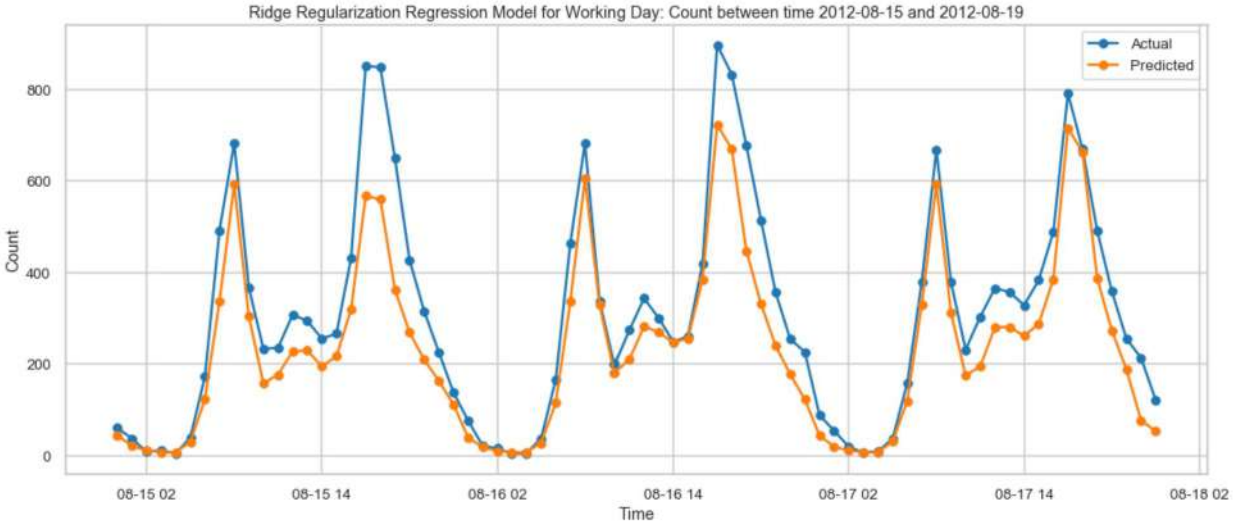
In [46]:
```
algo_score.loc['Ridge Regression', 'Training+Test Time (sec)'] = 1.3
cv_time.append(4.64)
```

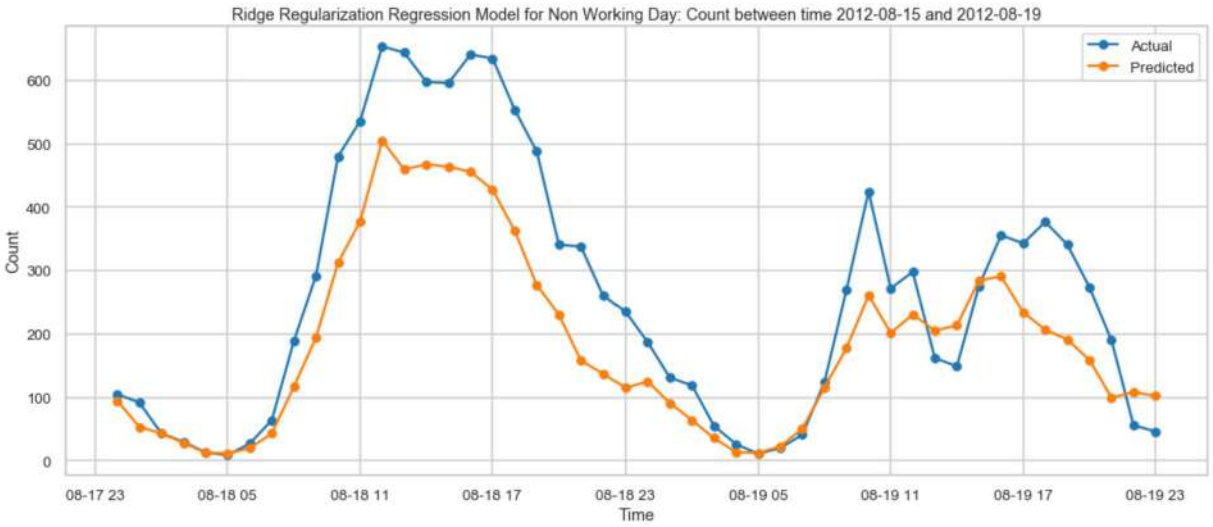
Obsevations

- Similar performance as Linear Regression.

Prediction Plot for Test data in a month (last 5 days of a month)

In [47]:
```
# Ridge Regression Plot: True vs. Predicted for one month
t_from, t_to = '2012-08-15', '2012-08-19'
ytest_w_predict, ytest_nw_predict = y_predict_summary[1], y_predict_summary[3]
ytest_w_predict = pd.Series(ytest_w_predict, index = ytest_w.index)
ytest_nw_predict = pd.Series(ytest_nw_predict, index = ytest_nw.index)

plot_true_vs_pred(ytest_w, ytest_nw, ytest_w_predict, ytest_nw_predict, 'Ridge Regular
```





Storing the regression coefficents for comparison later...

In [48]:
```
# Features and the Estimated Linear Regression Coefficients obtained for Working day d
df_coeff['Ridge_Coeff_Working'] = ridge_w.best_estimator_.coef_
df_coeff['Ridge_Coeff_Non_Working'] = ridge_nw.best_estimator_.coef_
```

Regularization Model - Lasso

Hyperparameter Tuning

In [49]:
```
# Lasso Regression
from sklearn.linear_model import Lasso
from sklearn.model_selection import GridSearchCV

# Hyperparameter Tuning
param_grid = {'alpha': [0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100]}
lasso_w = GridSearchCV(Lasso(random_state=42), param_grid, cv=5, scoring=rmsle_cv)
lasso_w.fit(X_w, logy_w)
print('Best alpha for Working Day Lasso Regression Model: {}'.format(lasso_w.best_para

lasso_nw = GridSearchCV(Lasso(random_state=42), param_grid, cv=5, scoring=rmsle_cv)
lasso_nw.fit(X_nw, logy_nw)
print('Best alpha for Non Working Day Lasso Regression Model: {}'.format(lasso_nw.best
```

```
Best alpha for Working Day Lasso Regression Model: {'alpha': 0.1}
Best alpha for Non Working Day Lasso Regression Model: {'alpha': 0.5}
```

Model Fit + Predict

In [50]:
```
param_summary = [lasso_w.best_params_, lasso_nw.best_params_,'']

rmsle_summary, y_predict_summary = model_fit(lasso_w, X_w, Xtest_w, y_w, ytest_w, lass
ypred_test.loc[Xtest.workingday==1,'Lasso'], ypred_test.loc[Xtest.workingday==0,'Lasso
```

In [51]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(lasso_w, X_w, y_w, lasso_nw, X_nw
ypred_train.loc[X.workingday==1,'Lasso'], ypred_train.loc[X.workingday==0,'Lasso'] = y

algo_score.loc['Lasso Regression'] = rmsle_summary+rmsle_val_summary+param_summary
algo_score.loc[['Lasso Regression']]
```

Out[51]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Val (A
Lasso Regression	1.313353	1.065252	1.241062	1.285346	1.117959	1.231794	1.31459	1.075364	1

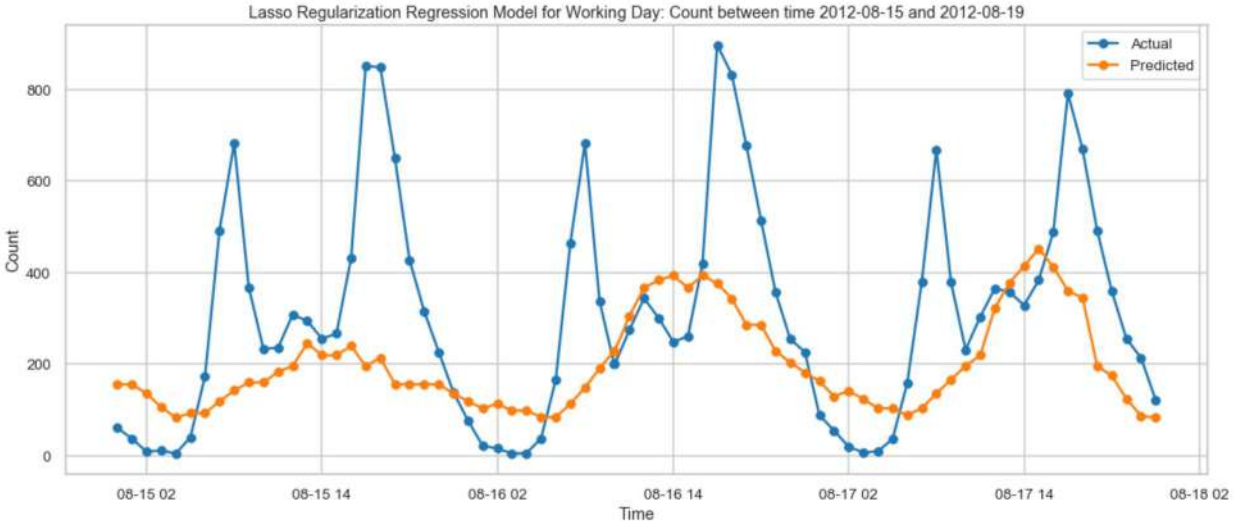
In [52]:
```
algo_score.loc['Lasso Regression', 'Training+Test Time (sec)'] = 1.3
cv_time.append(4.57)
```

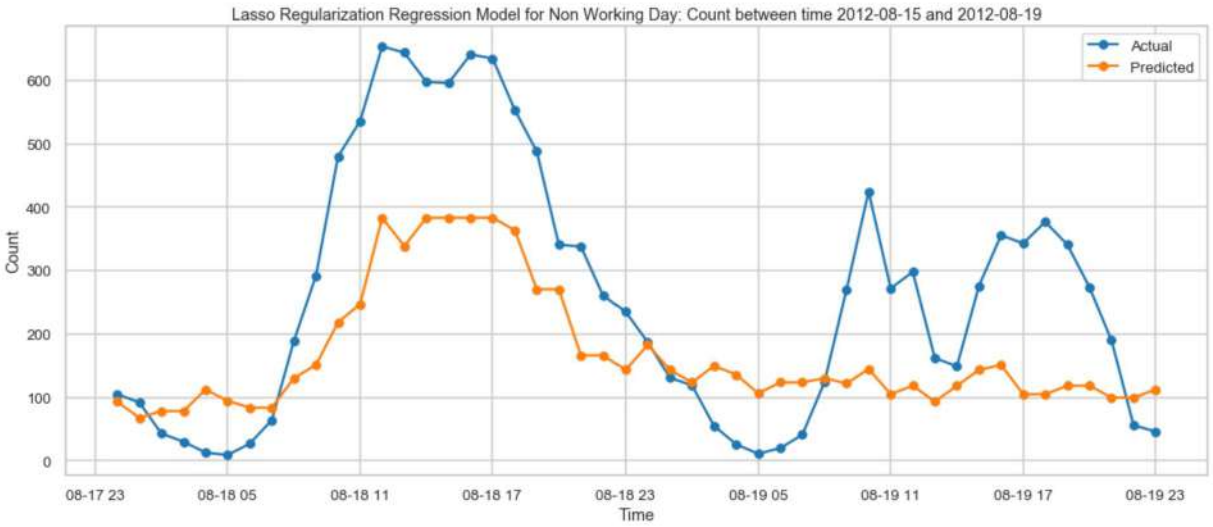
Observations

- Very poor performance. It probably zeros out several features

Prediction Plot for Test data in a month (last 5 days of a month)

In [53]:
```
# Lasso Regression Plot: True vs. Predicted for one month
t_from, t_to = '2012-08-15', '2012-08-19'
ytest_w_predict, ytest_nw_predict = y_predict_summary[1], y_predict_summary[3]
ytest_w_predict = pd.Series(ytest_w_predict, index = ytest_w.index)
ytest_nw_predict = pd.Series(ytest_nw_predict, index = ytest_nw.index)

plot_true_vs_pred(ytest_w, ytest_nw, ytest_w_predict, ytest_nw_predict, 'Lasso Regular
```





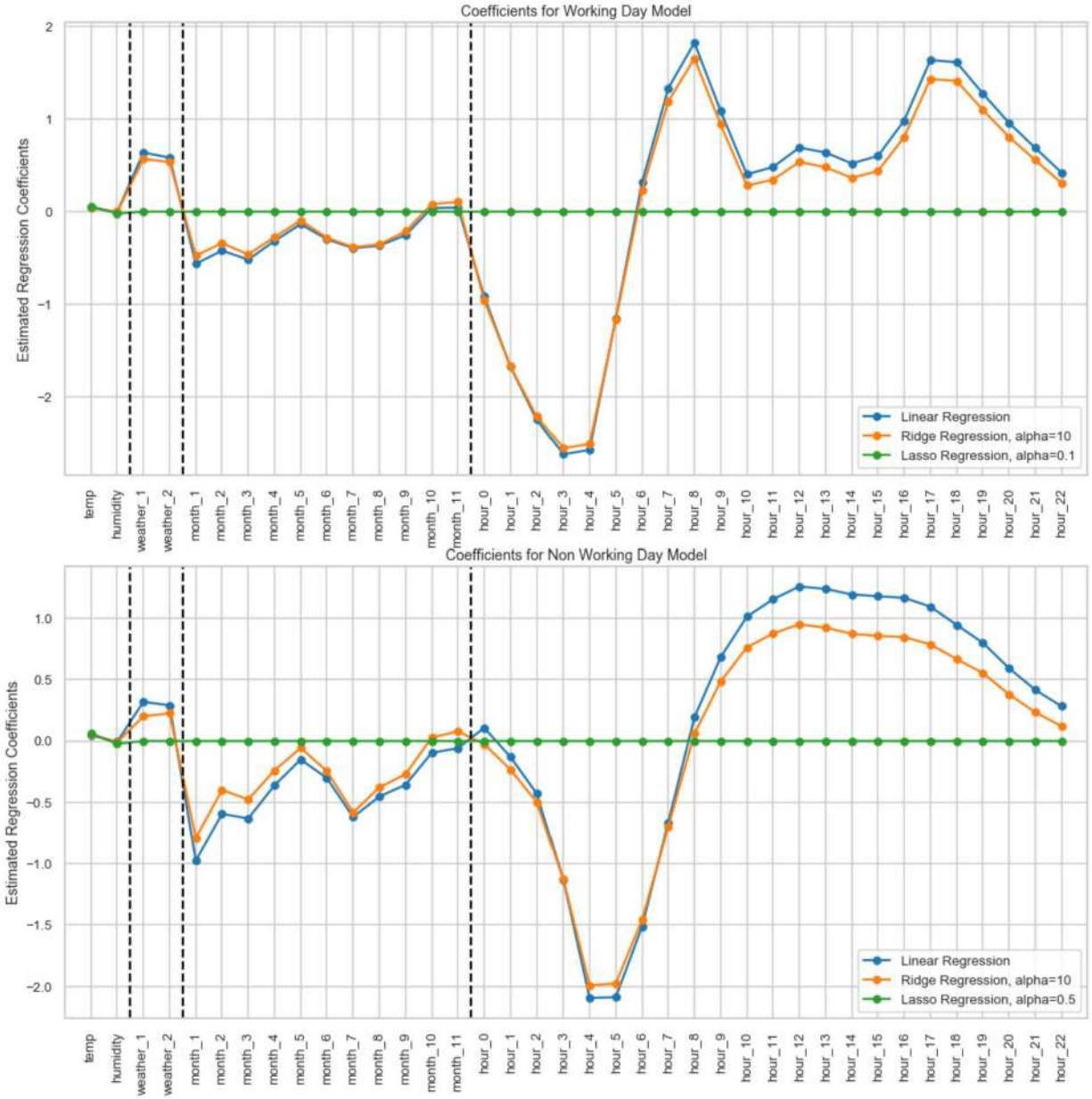
Storing the regression coefficents for comparison

In [54]:
```
# Features and the Estimated Linear Regression Coefficients obtained for Working day c
df_coeff['Lasso_Coeff_Working'] = lasso_w.best_estimator_.coef_
df_coeff['Lasso_Coeff_Non_Working'] = lasso_nw.best_estimator_.coef_
```

Feature Coefficient Comparison: Linear vs. Ridge vs. Lasso

In [55]:
```
# Plotting the feature coefficients for Linear, Ridge and Lasso Regression Models
fig = plt.figure(figsize=(18, 18))

# Working day plot
axes = fig.add_subplot(2, 1, 1)
axes.plot(df_coeff.Lin_Coeff_Working, label='Linear Regression', marker='.', markersiz
axes.plot(df_coeff.Ridge_Coeff_Working, label='Ridge Regression, alpha={}'.format(ridg
axes.plot(df_coeff.Lasso_Coeff_Working, label='Lasso Regression, alpha={}'.format(lass
axes.axvline(2-0.5, c='k', ls='--')
axes.axvline(4-0.5,  c='k', ls='--')
axes.axvline(15-0.5,  c='k', ls='--')
plt.xticks(range(len(df_coeff.Lin_Coeff_Working)), df_coeff.features, rotation=90)
axes.set(ylabel='Estimated Regression Coefficients', title='Coefficients for Working D
axes.set(xlim=[-1, len(df_coeff.Lin_Coeff_Working)])
axes.legend()
```

```
axes = fig.add_subplot(2, 1, 2)
axes.plot(df_coeff.Lin_Coeff_Non_Working, label='Linear Regression', marker='.', marke
axes.plot(df_coeff.Ridge_Coeff_Non_Working, label='Ridge Regression, alpha={}'.format(
axes.plot(df_coeff.Lasso_Coeff_Non_Working, label='Lasso Regression, alpha={}'.format(
axes.axvline(2-0.5,  c='k', ls='--')
axes.axvline(4-0.5,  c='k', ls='--')
axes.axvline(15-0.5,  c='k', ls='--')
plt.xticks(range(len(df_coeff.Lin_Coeff_Working)), df_coeff.features, rotation=90)
axes.set(ylabel='Estimated Regression Coefficients', title='Coefficients for Non Worki
axes.set(xlim=[-1, len(df_coeff.Lin_Coeff_Working)])
axes.legend()

plt.show()
```



Observations

- As expected from the minimizing function, Linear Regression coefficients are most agressive (highest magnitude). Ridge Regression coefficients closely follows Linear Regression. And Lasso more or less nullifies most of the coefficients.

- Lasso model provides a non-zero value only for temp and humidity for working day model
- From the Linear and Ridge coefficient plots, we can see that maximum dependency on the bike rental count are the hour in the day (highest coefficient magnitudes). Negative values of coefficient for early morning hours and positive values with greater magnitude during peak hours of the respective models.
  - Working day coefficent has highest positive coefficient values at 8am and 5pm
  - Non-working day coefficients has a single peak across hours ~ 12 noon (as observed from our earlier plots)
- Weather_1 and weather_2 have positive coefficient value, indicating a negative bias for weather_3 (light snow/rain). Note that if weather=3, then weather_1 and weather_2 = 0. Hence effective weather_3 coefficient = 0
- Though the coefficient value for temp and humidity is low compared to the others, the range of these values are higher too.
- The absolute value of the coefficients are higher for months 5, 6, 10, 11, 12 indicating higher bike rentals during that month.

Ensemble Model - Random Forest

Single Model for Working and Non-working days + Categorical Features

Since Random Forest Regression predicts based on decision tree, we expect it to handle categorical features including hours, months, working and non-working day gracefully. Let us start off by including those features into our model (instead of the using the transformed features + 2 models for working and non-working) and see its performance first. It would be a simple single model and easier to analyze

In [56]:
```
# Head of the Traning data
X2.head(n=3)
```

Out[56]:

datetime	workingday	weather	temp	humidity	month	hour
2011-01-01 00:00:00	0	1	9.84	81	1	0
2011-01-01 01:00:00	0	1	9.02	80	1	1
2011-01-01 02:00:00	0	1	9.02	80	1	2

Hyperparameter Tuning

Procedure - tuning 1 parameter at a time

1. First obtain n_estimators using default values of the remaining parameters
2. Tune for the max_features using the best n_estimators
3. Tune for min_samples_leaf using the best n_estimators and max_features
4. Tune for max_depth using the best n_estimators, max_features and min_samples_leaf

5. Tune for min_samples_split using the best n_estimators, max_features, min_samples_leaf and max_depth

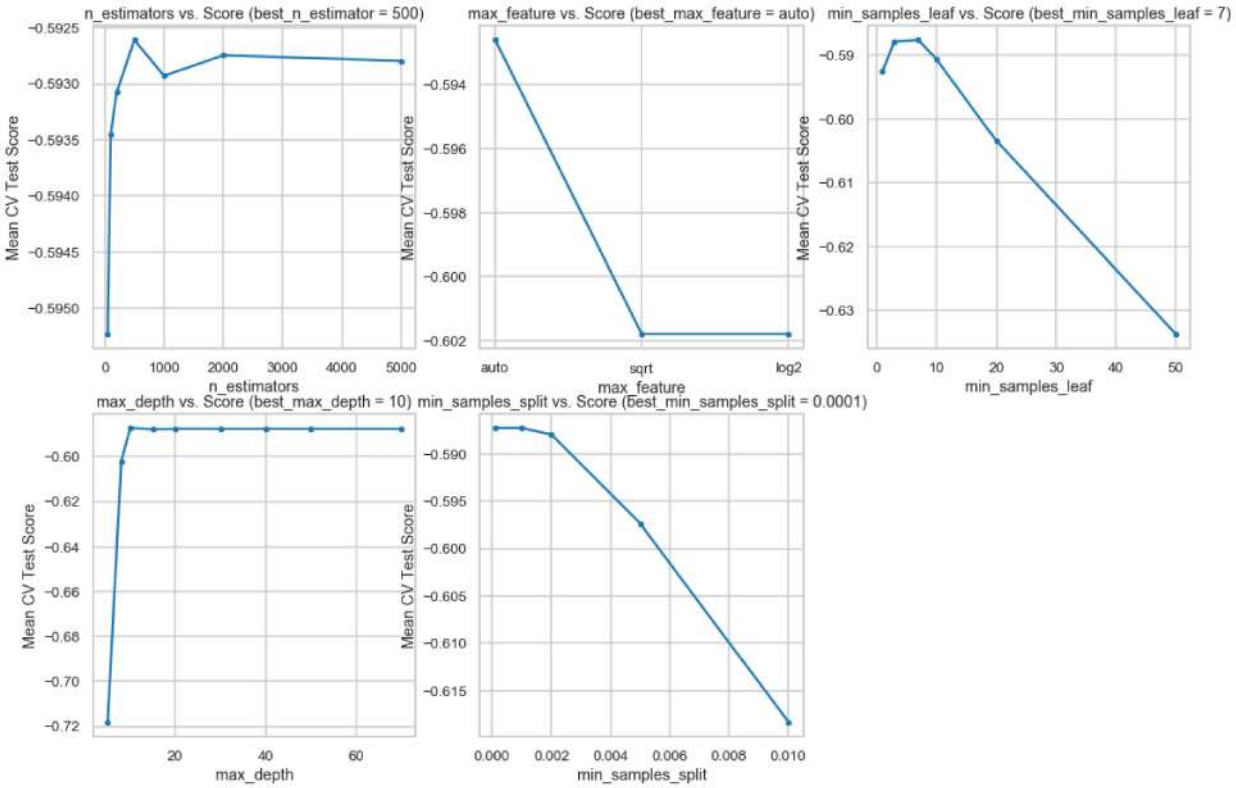
In [57]:
```
## Random Forest Regression Hyperparameter tuning using Grid Search to obtain the best
## Commented it out since it takes a lot of time to run. Using the best parameters obt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators': [50, 100, 200, 500, 1000, 2000, 5000]}
#rf_main = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scor
#rf_main.fit(X2, logy2)

#param_grid = {'n_estimators': [500], 'max_features':['auto', 'sqrt', 'log2']}
#rf_main = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scor
#rf_main.fit(X2, logy2)

#param_grid = {'n_estimators': [500], 'max_features':['auto'], 'min_samples_leaf':[1,
#rf_main = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scor
#rf_main.fit(X2, logy2)

#param_grid = {'n_estimators': [500], 'max_features':['auto'], 'min_samples_leaf':[7],
#rf_main = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scor
#rf_main.fit(X2, logy2)

#param_grid = {'n_estimators': [500], 'max_features':['auto'], 'min_samples_leaf':[7],
#rf_main = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scor
#rf_main.fit(X2, logy2)

#print('Best parameters for Random Forest Regression Model: {}'.format(rf_main.best_pa
```

```
C:\Users\smaiya\AppData\Local\Continuum\anaconda3\lib\site-packages\sklearn\ensemble
\weight_boosting.py:29: DeprecationWarning: numpy.core.umath_tests is an internal Num
Py module and should not be imported. It will be removed in a future NumPy release.
  from numpy.core.umath_tests import inner1d
```

Hyperparameter Tuning Score Plot

In [58]:
```
# All the below results are obtained from the above GridSearchCV hyperparamter tuning
fig=plt.figure(figsize=(18, 12))

n_est_array = [50, 100, 200, 500, 1000, 2000, 5000]
n_est_cv_score = [-0.59522862, -0.59345376, -0.59307606, -0.59260841, -0.59292831,-0.5
axes=fig.add_subplot(2, 3, 1)
axes.plot(n_est_array, n_est_cv_score, marker='.')
axes.set(xlabel='n_estimators', ylabel='Mean CV Test Score', title='n_estimators vs. S

max_feature_array = ['auto', 'sqrt', 'log2']
max_feature_cv_score = [-0.59260841, -0.60178303, -0.60178303]
axes=fig.add_subplot(2, 3, 2)
axes.plot(range(3), max_feature_cv_score, marker='.')
plt.xticks(range(3), max_feature_array)
axes.set(xlabel='max_feature', ylabel='Mean CV Test Score', title='max_feature vs. Sco

min_samples_leaf_array = [1, 3, 7, 10, 20, 50]
min_samples_leaf_cv_score = [-0.59260841, -0.58788348, -0.58764415, -0.59060404, -0.60
axes=fig.add_subplot(2, 3, 3)
axes.plot(min_samples_leaf_array, min_samples_leaf_cv_score, marker='.')
axes.set(xlabel='min_samples_leaf', ylabel='Mean CV Test Score', title='min_samples_le
```

```
max_depth_array = [5, 8, 10, 15, 20, 30, 40, 50, 70]
max_depth_cv_score = [-0.71808668, -0.60209726, -0.58727382, -0.58785334, -0.58764068,
axes=fig.add_subplot(2, 3, 4)
axes.plot(max_depth_array, max_depth_cv_score, marker='.')
axes.set(xlabel='max_depth', ylabel='Mean CV Test Score', title='max_depth vs. Score (

min_samples_split_array = [0.0001, 0.001, 0.002, 0.005, 0.01]
min_samples_split_cv_score = [-0.58727382, -0.58727382, -0.58796633, -0.59735637, -0.6
axes=fig.add_subplot(2, 3, 5)
axes.plot(min_samples_split_array, min_samples_split_cv_score, marker='.')
axes.set(xlabel='min_samples_split', ylabel='Mean CV Test Score', title='min_samples_s

plt.show()
```



Model Fit + Predict

In [59]:
```
# Random Forest Regression
from sklearn.ensemble import RandomForestRegressor

# Best parameters obtained via GridSearchCV above
best_n_estimators, best_max_features = 500, 'auto'
best_min_samples_leaf, best_max_depth = 7, 10
param_summary = ['n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:
                 'n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:

rfa = RandomForestRegressor(n_estimators = best_n_estimators, max_features = best_max_
                            min_samples_leaf = best_min_samples_leaf, max_depth = best_

rmsle_summary, y_predict_summary = model_fit(rfa, X2, Xtest2, y2, ytest2)
ypred_test['RF1'] = y_predict_summary[1]
```

In [60]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(rfa, X2, y2)
ypred_train['RF1'] = y_predict_val_summary[0]
```

```
algo_score.loc['Random Forest-Categorical+Single'] = rmsle_summary+rmsle_val_summary+p
algo_score.loc[['Random Forest-Categorical+Single']]
```

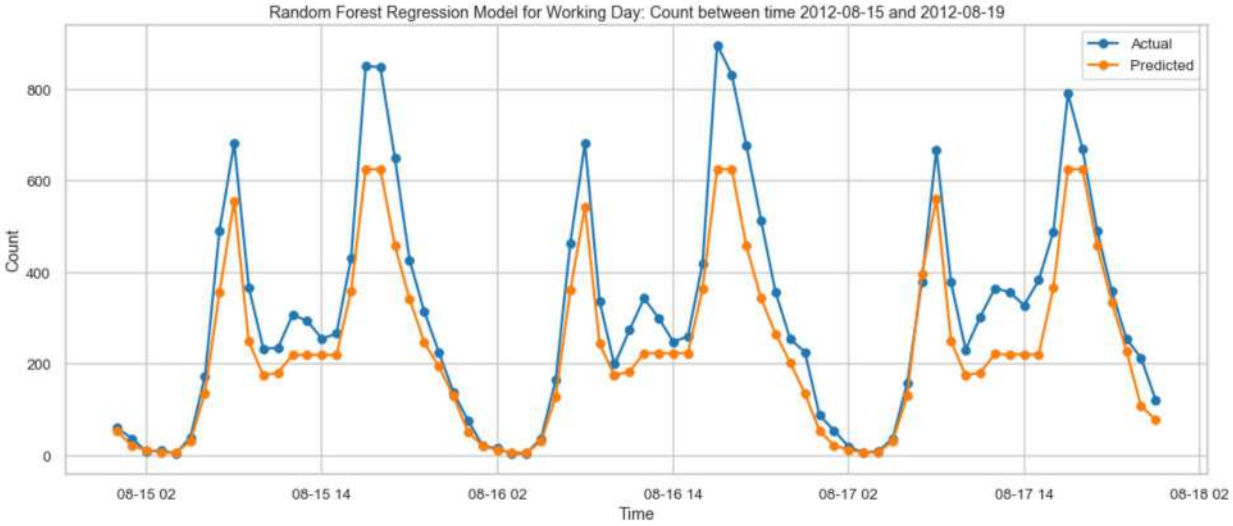
Out[60]:

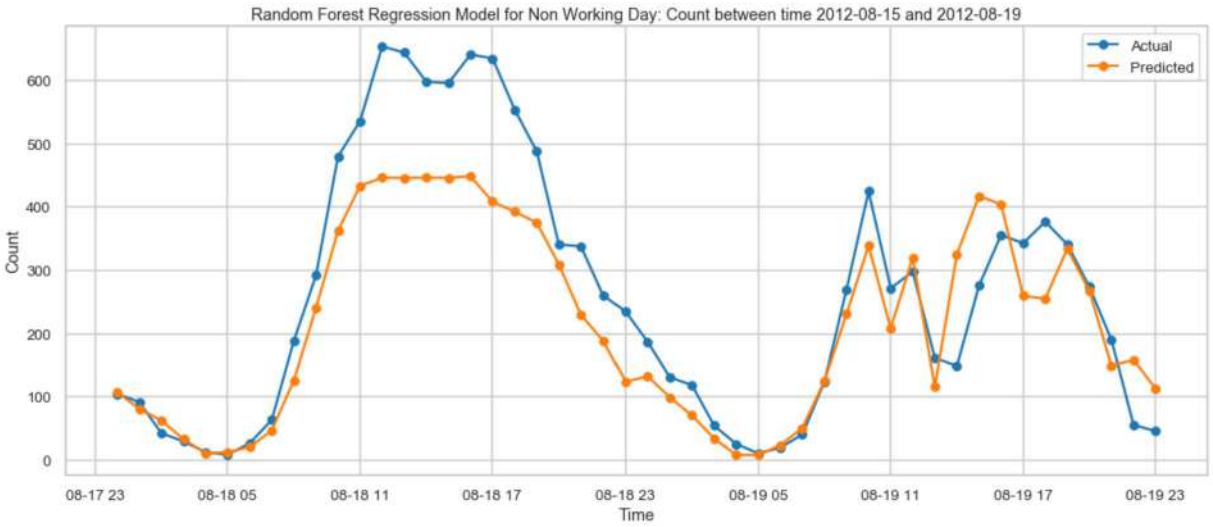
Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validat RM (N Work D
Random Forest-Categorical+Single	0.353823	0.387612	0.364732	0.393007	0.48925	0.427676	0.425339	0.482

In [61]:
```
algo_score.loc['Random Forest-Categorical+Single', 'Training+Test Time (sec)'] = 5.48
cv_time.append(19.9)
```

Prediction Plot for Test data in a month (last 5 days of a month)

In [62]:
```
# Random Forest Regression Plot: True vs. Predicted for one month
t_from, t_to = '2012-08-15', '2012-08-19'
y_test_predict =  pd.Series(y_predict_summary[1], index = ytest2.index)
ytest_w_predict, ytest_nw_predict = y_test_predict[Xtest2.workingday==1], y_test_predi
plot_true_vs_pred(ytest2_w, ytest2_nw, ytest_w_predict, ytest_nw_predict, 'Random Fore
```





We notice that there are some irregularities with the predictions on 8/19 compared to 8/18 from the non-working day predicted data. Let us take a look at the data for these days to see if the model predictions makes sense

In [63]:
```
Xtest2['2012-08-19 09':'2012-08-19 15']
```

Out[63]:

datetime	workingday	weather	temp	humidity	month	hour
2012-08-19 09:00:00	0	2	27.06	74	8	9
2012-08-19 10:00:00	0	2	27.88	69	8	10
2012-08-19 11:00:00	0	3	26.24	78	8	11
2012-08-19 12:00:00	0	3	26.24	73	8	12
2012-08-19 13:00:00	0	3	26.24	83	8	13
2012-08-19 14:00:00	0	3	26.24	73	8	14
2012-08-19 15:00:00	0	2	26.24	65	8	15

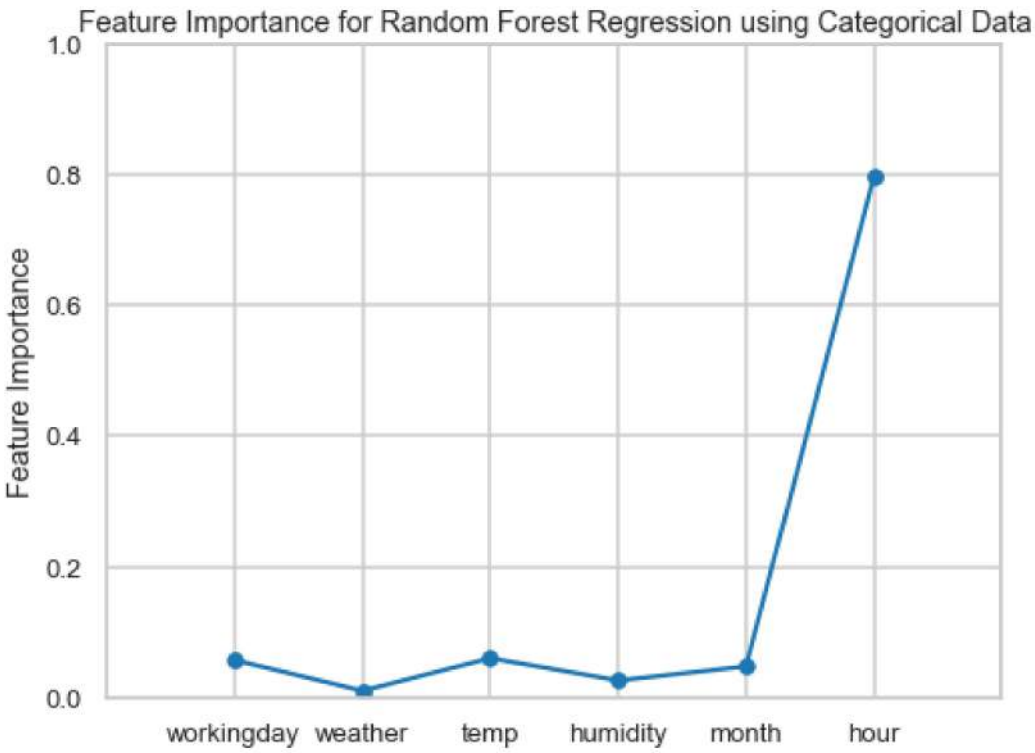
From the above table, we notice that the weather gets bad between 11:00 and 14:00 on the 19th. As a result, the model predicts a lower count value for those hours.

Observations

- We see marginal gain (Test RMSLE reduced to 0.43 from 0.44) over the linear models

Feature Importance Plot

In [64]:
```
# Plotting the Feature Importance
fig = plt.figure(figsize=(8, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(rfa.feature_importances_, marker='.', markersize=15)
plt.xticks(range(len(rfa.feature_importances_)), X2.columns)
axes.set(ylabel='Feature Importance', title='Feature Importance for Random Forest Regr
axes.set(xlim=[-1, len(X2.columns)], ylim=[0, 1])

plt.show()
```



Observations

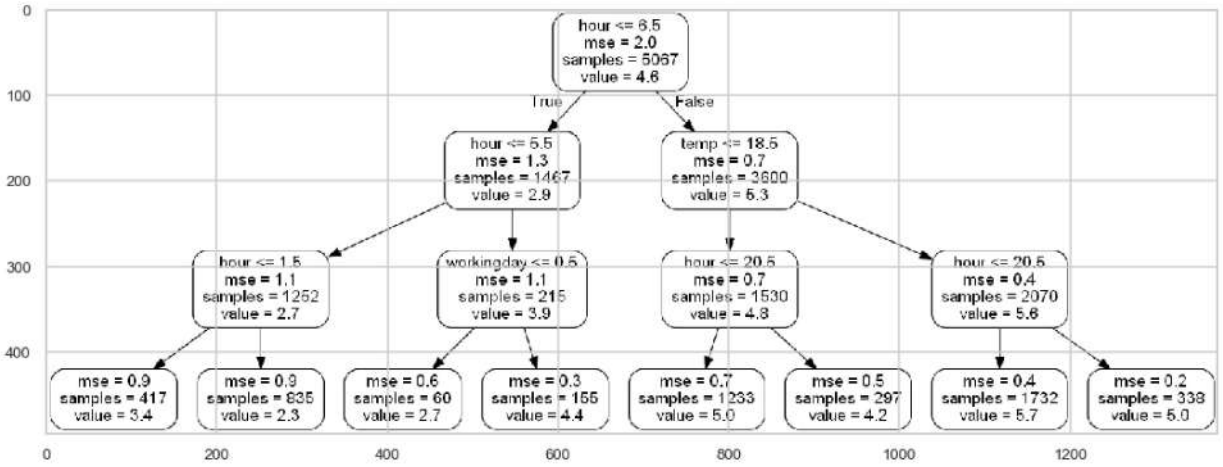
- As expected 'hour' feature holds maximum importance. We saw spikes and dips in count value depending on the hour of the day
- We notice that workingday feature has marginal importance

Let us obtain 2 separate models for working day and non-working day

In [65]:
```
# Visualizing a graph limiting the tree to 3 levels to gain some understanding
rf_small = RandomForestRegressor(n_estimators=10, max_features = 'auto', max_depth=3,
rf_small.fit(X2, logy2)
logy2_predict_small = rf_small.predict(X2)
```

```
rmsle2_small = rmsle(y2, np.expm1(logy2_predict_small))
print('RMSLE score for Random Forest Regressor using a small tree (max_depth = 3) for

# Import tools needed for visualization
from sklearn.tree import export_graphviz
import pydot
import matplotlib.image as mpimg

# Pull out one tree from the forest
tree = rf_small.estimators_[0]
# Export the image to a dot file
dot_data = export_graphviz(tree, out_file = './Images/tree_rf1.dot', feature_names = )
# Use dot file to create a graph
(graph, ) = pydot.graph_from_dot_file('./Images/tree_rf1.dot')
# Write graph to a png file
graph.write_png('./Images/tree_rf1.png')

img=mpimg.imread('./Images/tree_rf1.png')
plt.figure(figsize=(18, 12))
imgplot = plt.imshow(img)
plt.show()
```

```
RMSLE score for Random Forest Regressor using a small tree (max_depth = 3) for simple
visualization = 0.734
```



Two Separate Models + Binary Vector Features via OneHotEncoder

Now let model using the other extreme case -

- All categorical features (weather, month and hour) split into Binary Vector Features, namely weather_1, weather_2, month_1, month_2, ..., month_11, hour_1, hour_2, ..., hour_22,
- Use two separate models for working day and non-working day to predict the bike count

Hyperparameter Tuning

Procedure

1. Tune for (n_estimators, max_features) using default values of the remaining parameters
2. Tune for min_samples_leaf using the best n_estimators and max_features
3. Tune for max_depth using the best n_estimators, max_features and min_samples_leaf

4. Tune for min_samples_split using the best n_estimators, max_features, min_samples_leaf and max_depth

In [66]:
```
# Head of the Traning data
X_w.head(n=2)
```

Out[66]:

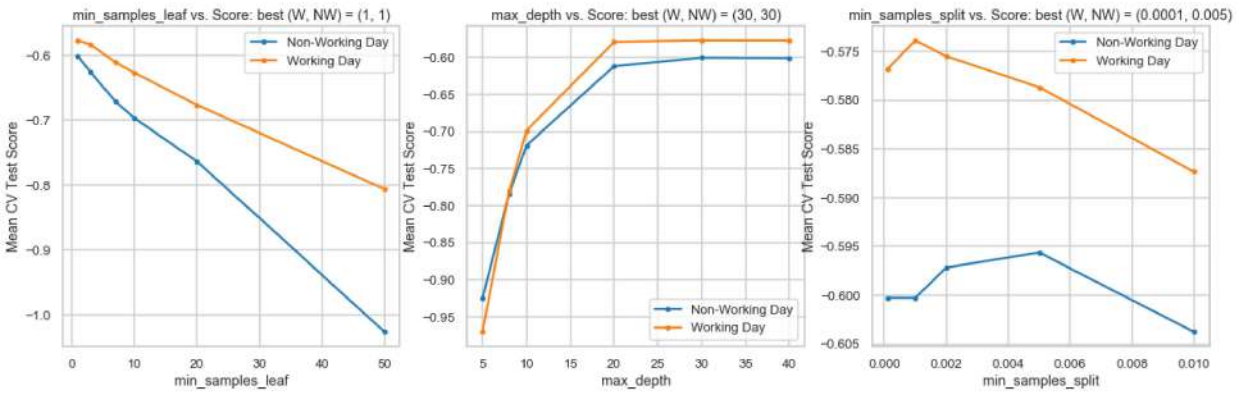
datetime	temp	humidity	weather_1	weather_2	month_1	month_2	month_3	month_4	month_5
2011-01-03 00:00:00	9.02	44	1	0	1	0	0	0	0
2011-01-03 01:00:00	8.20	44	1	0	1	0	0	0	0

2 rows × 38 columns

In [67]:
```
## Random Forest Regression Grid Search to obtain the best parameters.
## Commented it out since it takes a lot of time to run. Using the best parameters obt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators': [50, 100, 200, 500, 1000, 2000, 5000], 'max_features':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scoring
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scorir
#rf_w.fit(X_w, logy_w)
#rf_nw.fit(X_nw, logy_nw)

#param_grid_w = {'n_estimators': [2000], 'max_features':['sqrt'], 'min_samples_leaf':[
#param_grid_nw = {'n_estimators': [200], 'max_features':['log2'], 'min_samples_leaf':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_w, cv=5, scori
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_nw, cv=5, scc
#rf_w.fit(X_w, logy_w)
#rf_nw.fit(X_nw, logy_nw)

#param_grid_w = {'n_estimators': [2000], 'max_features':['sqrt'], 'min_samples_leaf':[
#param_grid_nw = {'n_estimators': [200], 'max_features':['log2'], 'min_samples_leaf':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_w, cv=5, scori
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_nw, cv=5, scc
#rf_w.fit(X_w, logy_w)
#rf_nw.fit(X_nw, logy_nw)

#param_grid_w = {'n_estimators': [2000], 'max_features':['sqrt'], 'min_samples_leaf':[
#param_grid_nw = {'n_estimators': [200], 'max_features':['log2'], 'min_samples_leaf':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_w, cv=5, scori
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_nw, cv=5, scc
#rf_w.fit(X_w, logy_w)
#rf_nw.fit(X_nw, logy_nw)

#print('Best parameters for Random Forest Regression Model for Working days: {}'.forma
#print('Best parameters for Random Forest Regression Model for Non Working days: {}'.f
```

Hyperparameter Tuning Score Plots

In [68]:
```
# All the below results are obtained from the above GridSearchCV hyperparamter tuning

fig=plt.figure(figsize=(21, 6))

min_samples_leaf_array = [1, 3, 7, 10, 20, 50]
min_samples_leaf_cv_score_nw = [-0.6014166 , -0.62586217, -0.6713266 , -0.69684692, -0
min_samples_leaf_cv_score_w = [-0.57692764, -0.58339873, -0.61078536, -0.62668656, -0.
axes=fig.add_subplot(1, 3, 1)
axes.plot(min_samples_leaf_array, min_samples_leaf_cv_score_nw, marker='.', label='Nor
axes.plot(min_samples_leaf_array, min_samples_leaf_cv_score_w, marker='.', label='Work
axes.set(xlabel='min_samples_leaf', ylabel='Mean CV Test Score', title='min_samples_le
axes.legend()

max_depth_array = [5, 8, 10, 20, 30, 40]
max_depth_cv_score_nw = [-0.92512009, -0.7837481 , -0.71864241, -0.61170644, -0.600258
max_depth_cv_score_w = [-0.97058161, -0.78030942, -0.69859085, -0.57880264, -0.576817
axes=fig.add_subplot(1, 3, 2)
axes.plot(max_depth_array, max_depth_cv_score_nw, marker='.', label='Non-Working Day')
axes.plot(max_depth_array, max_depth_cv_score_w, marker='.', label='Working Day')
axes.set(xlabel='max_depth', ylabel='Mean CV Test Score', title='max_depth vs. Score:
axes.legend()

min_samples_split_array = [0.0001, 0.001, 0.002, 0.005, 0.01]
min_samples_split_cv_score_nw = [-0.60025831, -0.6002647 , -0.59716144, -0.59562094, -
min_samples_split_cv_score_w = [-0.57681758, -0.57386935, -0.57549636, -0.57868933, -0
axes=fig.add_subplot(1, 3, 3)
axes.plot(min_samples_split_array, min_samples_split_cv_score_nw, marker='.', label='N
axes.plot(min_samples_split_array, min_samples_split_cv_score_w, marker='.', label='Wo
axes.set(xlabel='min_samples_split', ylabel='Mean CV Test Score', title='min_samples_s
axes.legend()

plt.show()
```



Model Fit + Predict

In [69]:
```
# Random Forest Regression

# Best parameters obtained via GridSearchCV above
best_n_estimators_w, best_max_features_w, best_min_samples_leaf_w, best_max_depth_w, b
best_n_estimators_nw, best_max_features_nw ,best_min_samples_leaf_nw, best_max_depth_n
param_summary = ['n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:
                 'n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:

print('Best parameters via GridSearchCV for Working Day:     '+param_summary[0])
print('Best parameters via GridSearchCV for Non Working Day: '+param_summary[1])
```

```
rfb_w = RandomForestRegressor(n_estimators=best_n_estimators_w, max_features=best_max_
                              max_depth=best_max_depth_w, min_samples_split=best_min_sa
rfb_nw = RandomForestRegressor(n_estimators=best_n_estimators_nw, max_features=best_ma
                              max_depth=best_max_depth_nw, min_samples_split=best_min_s

rmsle_summary, y_predict_summary = model_fit(rfb_w, X_w, Xtest_w, y_w, ytest_w, rfb_nw
ypred_test.loc[Xtest.workingday==1,'RF2'], ypred_test.loc[Xtest.workingday==0,'RF2'] =
```

```
Best parameters via GridSearchCV for Working Day:     n_estimators: 2000, max_feature
s: sqrt, min_samples_leaf: 1, max_depth: 30, min_samples_split: 0.0001
Best parameters via GridSearchCV for Non Working Day: n_estimators: 200, max_feature
s: log2, min_samples_leaf: 1, max_depth: 30, min_samples_split: 0.005
```

In [70]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(rfb_w, X_w, y_w, rfb_nw, X_nw, y_
ypred_train.loc[X.workingday==1,'RF2'], ypred_train.loc[X.workingday==0,'RF2'] = y_pre

algo_score.loc['Random Forest-OneHotEncoding'] = rmsle_summary+rmsle_val_summary+paran
algo_score.loc[['Random Forest-OneHotEncoding']]
```

Out[70]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validatio RMSL (No Workin Day
Random Forest-OneHotEncoding	0.181652	0.378901	0.259992	0.40938	0.541029	0.457731	0.437254	0.5326

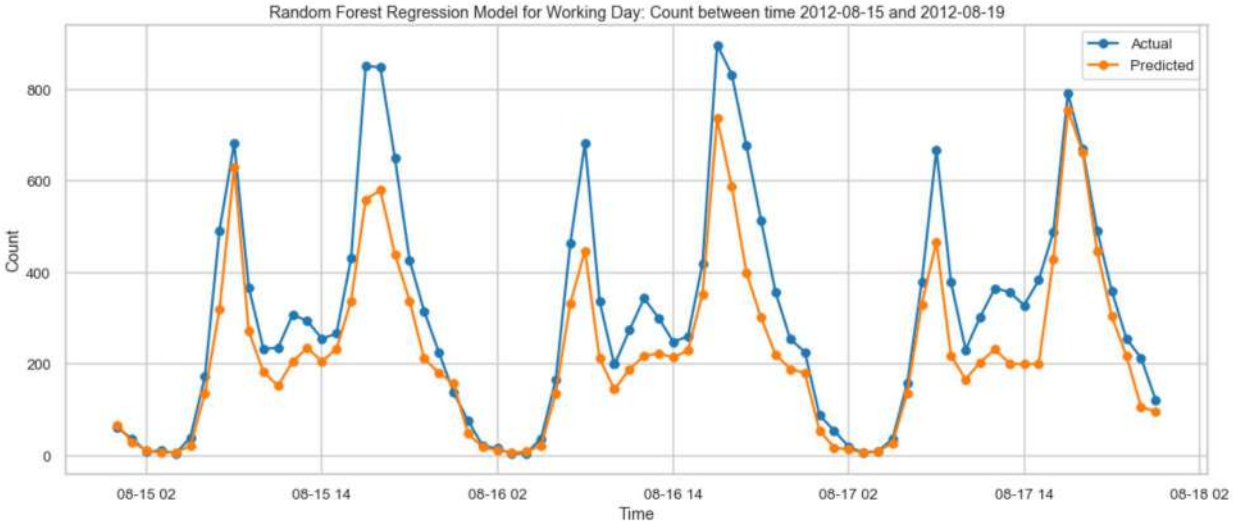
In [71]:
```
algo_score.loc['Random Forest-OneHotEncoding', 'Training+Test Time (sec)'] = 22.4
cv_time.append(80.3)
```

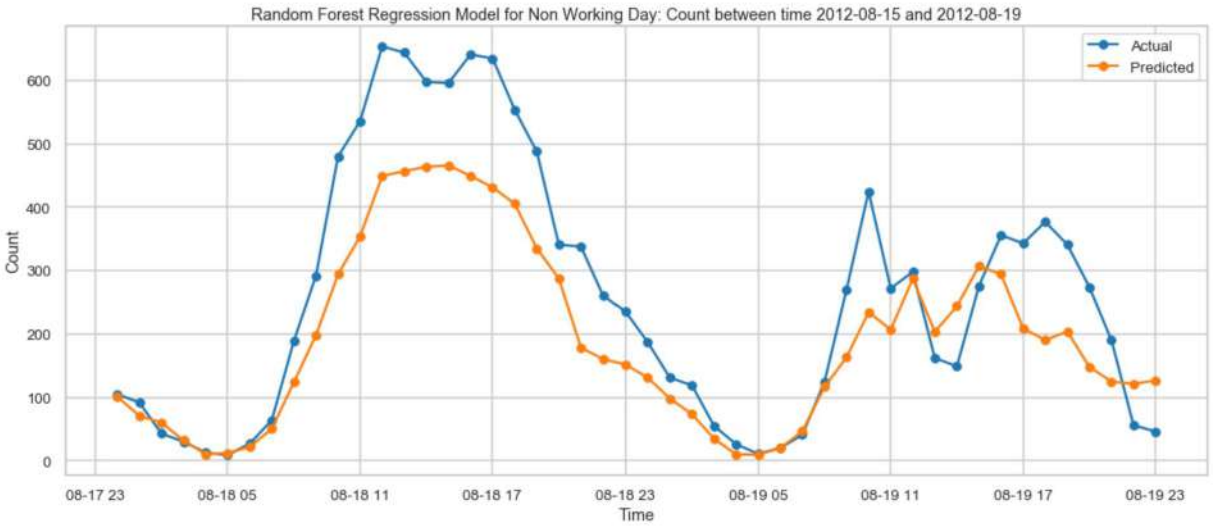
Observations

- We see a dip in prediction relative to our earlier Random Forest model (which used Categorical Features and single model for working and non-working days). Average Test RMSLE increases from 0.43 to 0.46
- Also note that the Train RMSLE is reasonably lower than Test RMSLE. This possibly is a result of on over-fit model

Prediction Plot for Test data in a month (last 5 days of a month)

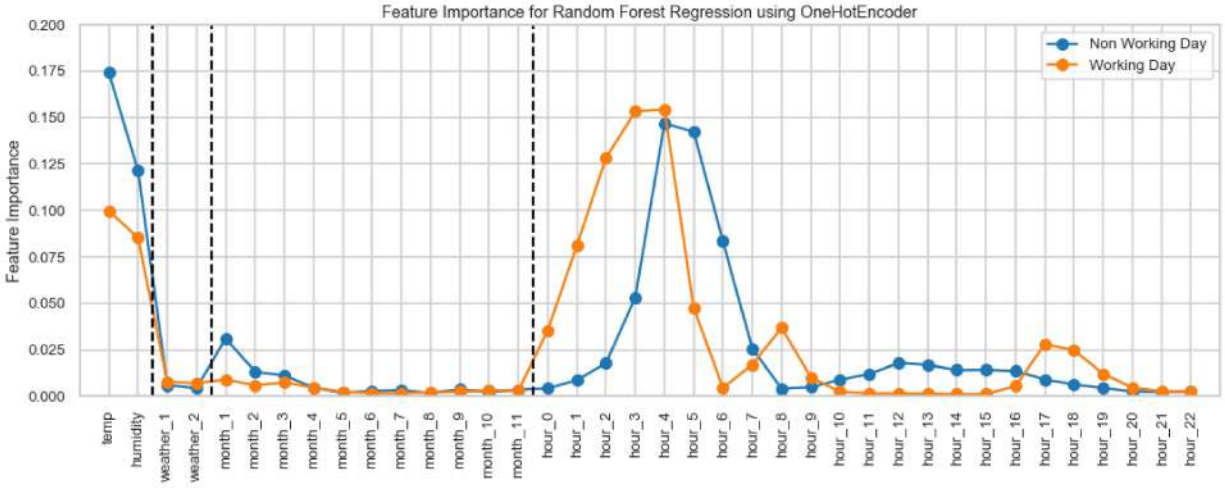
In [72]:
```
# Random Forest Regression Plot: True vs. Predicted for one month
t_from, t_to = '2012-08-15', '2012-08-19'
ytest_w_predict, ytest_nw_predict = y_predict_summary[1], y_predict_summary[3]
ytest_w_predict = pd.Series(ytest_w_predict, index = ytest_w.index)
ytest_nw_predict = pd.Series(ytest_nw_predict, index = ytest_nw.index)

plot_true_vs_pred(ytest_w, ytest_nw, ytest_w_predict, ytest_nw_predict, 'Random Forest
```





Feature Importance

In [73]:
```
# Plotting the Feature Importance
fig = plt.figure(figsize=(18, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(rfb_nw.feature_importances_, label='Non Working Day', marker='.', markersize
axes.plot(rfb_w.feature_importances_, label='Working Day', marker='.', markersize=20)
plt.xticks(range(len(rfb_w.feature_importances_)), X_w.columns, rotation=90)
axes.axvline(2-0.5, c='k', ls='--')
axes.axvline(4-0.5,  c='k', ls='--')
axes.axvline(15-0.5,  c='k', ls='--')
axes.set(ylabel='Feature Importance', title='Feature Importance for Random Forest Regr
axes.set(xlim=[-1, len(X_w.columns)], ylim=[0, 0.2])
axes.legend()

plt.show()
```



Observations

- Features for which the average count is significantly different from the generic mean count (late night or peak hours), tends to hold higher weightage. For example
  - Non working day hour_x feature set: hour_3 to hour_6 have high importance as the average count at these hours are very low compared to the daily average. Not splitting based on these features correctly would result in higher mse. The peak hours for non-working days are at 1pm and 2pm; correspondingly, we do see relatively higher importance for hour_12 and hour_13
  - Working day hour_x feature set: hour_1 to hour_4 have high importance for similar reasons explained in the previous bullet. hour_8, hour_17, hour_18 too hold reasonably high importance since they are the peak hours and tend to have higher than usual count values

Two Separate Models for Working and Non-working days + Categorical Features

- Let us try out an intermediate approach. Use 2 separate models for working and non-working days; but use the orginal categorical weather, month and hour data instead of the column vectors obtained using OneHotEncoder

Hyperparameter Tuning

Procedure

1. Tune for (n_estimators, max_features) using default values of the remaining parameters
2. Tune for min_samples_leaf using the best n_estimators and max_features
3. Tune for max_depth using the best n_estimators, max_features and min_samples_leaf
4. Tune for min_samples_split using the best n_estimators, max_features, min_samples_leaf and max_depth

In [74]:
```
# Head of the Traning data
X2_w.head(n=3)
```

Out[74]:

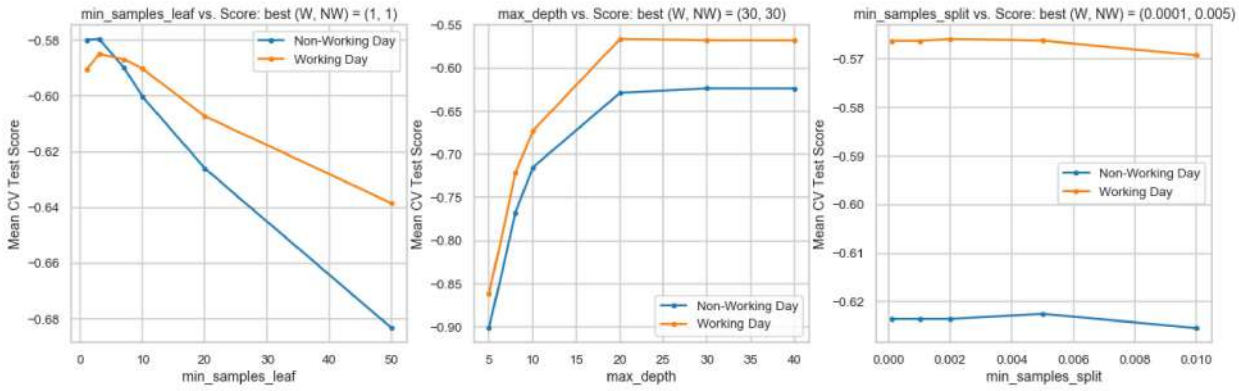
datetime	weather	temp	humidity	month	hour
2011-01-03 00:00:00	1	9.02	44	1	0
2011-01-03 01:00:00	1	8.20	44	1	1
2011-01-03 04:00:00	1	6.56	47	1	4

In [75]:
```
## Random Forest Regression Grid Search to obtain the best parameters.
## Commented it out since it takes a lot of time to run. Using the best parameters obt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators': [50, 100, 200, 500, 1000, 2000, 5000], 'max_features':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scoring
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scorir
#rf_w.fit(X_w, logy_w)
#rf_nw.fit(X_nw, logy_nw)

#param_grid_w = {'n_estimators': [500], 'max_features':['auto'], 'min_samples_leaf':[1
#param_grid_nw = {'n_estimators': [500], 'max_features':['sqrt'], 'min_samples_leaf':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_w, cv=5, scori
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_nw, cv=5, scc
#rf_w.fit(X2_w, logy2_w)
#rf_nw.fit(X2_nw, logy2_nw)

#param_grid_w = {'n_estimators': [500], 'max_features':['auto'], 'min_samples_leaf':[3
#param_grid_nw = {'n_estimators': [500], 'max_features':['sqrt'], 'min_samples_leaf':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_w, cv=5, scori
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_nw, cv=5, scc
#rf_w.fit(X_w, logy_w)
#rf_nw.fit(X_nw, logy_nw)

#param_grid_w = {'n_estimators': [500], 'max_features':['auto'], 'min_samples_leaf':[3
#param_grid_nw = {'n_estimators': [500], 'max_features':['sqrt'], 'min_samples_leaf':[
#rf_w = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_w, cv=5, scori
#rf_nw = GridSearchCV(RandomForestRegressor(random_state=42), param_grid_nw, cv=5, scc
#rf_w.fit(X_w, logy_w)
#rf_nw.fit(X_nw, logy_nw)

#print('Best parameters for Random Forest Regression Model for Working days: {}'.formc
#print('Best parameters for Random Forest Regression Model for Non Working days: {}'.f
```

Hyperparameter Tuning Score Plots

In [76]:
```
# All the below results are obtained from the above GridSearchCV hyperparamter tuning

fig=plt.figure(figsize=(21, 6))

min_samples_leaf_array = [1, 3, 7, 10, 20, 50]
min_samples_leaf_cv_score_nw = [-0.57970417, -0.57946332, -0.58976635, -0.6002362 , -0
       -0.68333543]
min_samples_leaf_cv_score_w = [-0.59039957, -0.58484149, -0.58673642, -0.59015947, -0.
       -0.63850817]
axes=fig.add_subplot(1, 3, 1)
axes.plot(min_samples_leaf_array, min_samples_leaf_cv_score_nw, marker='.', label='Nor
axes.plot(min_samples_leaf_array, min_samples_leaf_cv_score_w, marker='.', label='Work
```

```
axes.set(xlabel='min_samples_leaf', ylabel='Mean CV Test Score', title='min_samples_le
axes.legend()

max_depth_array = [5, 8, 10, 20, 30, 40]
max_depth_cv_score_nw = [-0.9013261 , -0.76711388, -0.7151308 , -0.6287906 , -0.623546
max_depth_cv_score_w = [-0.86157536, -0.72155214, -0.67269418, -0.56630213, -0.567773
axes=fig.add_subplot(1, 3, 2)
axes.plot(max_depth_array, max_depth_cv_score_nw, marker='.', label='Non-Working Day')
axes.plot(max_depth_array, max_depth_cv_score_w, marker='.', label='Working Day')
axes.set(xlabel='max_depth', ylabel='Mean CV Test Score', title='max_depth vs. Score:
axes.legend()

min_samples_split_array = [0.0001, 0.001, 0.002, 0.005, 0.01]
min_samples_split_cv_score_nw = [-0.62354017, -0.62354017, -0.62354017, -0.62250308, -
min_samples_split_cv_score_w = [-0.56630213, -0.56630213, -0.56592649, -0.56624378, -0
axes=fig.add_subplot(1, 3, 3)
axes.plot(min_samples_split_array, min_samples_split_cv_score_nw, marker='.', label='N
axes.plot(min_samples_split_array, min_samples_split_cv_score_w, marker='.', label='Wo
axes.set(xlabel='min_samples_split', ylabel='Mean CV Test Score', title='min_samples_s
axes.legend()

plt.show()
```



Model Fit + Predict

In [77]:
```
# Random Forest Regression

# Best parameters obtained via GridSearchCV above
best_n_estimators_w, best_max_features_w, best_min_samples_leaf_w, best_max_depth_w, b
best_n_estimators_nw, best_max_features_nw ,best_min_samples_leaf_nw, best_max_depth_n
param_summary = ['n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:
                 'n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:

print('Best parameters via GridSearchCV for Working Day:     '+param_summary[0])
print('Best parameters via GridSearchCV for Non Working Day: '+param_summary[1])

rfc_w = RandomForestRegressor(n_estimators=best_n_estimators_w, max_features=best_max_
                              max_depth=best_max_depth_w, min_samples_split=best_min_sa
rfc_nw = RandomForestRegressor(n_estimators=best_n_estimators_nw, max_features=best_ma
                               max_depth=best_max_depth_nw, min_samples_split=best_min_s

rmsle_summary, y_predict_summary = model_fit(rfc_w, X2_w, Xtest2_w, y2_w, ytest2_w, rf
ypred_test.loc[Xtest2.workingday==1,'RF3'], ypred_test.loc[Xtest2.workingday==0,'RF3']
```

```
Best parameters via GridSearchCV for Working Day:     n_estimators: 500, max_feature
s: auto, min_samples_leaf: 3, max_depth: 20, min_samples_split: 0.002
Best parameters via GridSearchCV for Non Working Day: n_estimators: 500, max_feature
s: sqrt, min_samples_leaf: 3, max_depth: 30, min_samples_split: 0.005
```

In [78]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(rfc_w, X2_w, y2_w, rfc_nw, X2_nw,
ypred_train.loc[X2.workingday==1,'RF3'], ypred_train.loc[X2.workingday==0,'RF3'] = y_p

algo_score.loc['Random Forest-Categorical Features'] = rmsle_summary+rmsle_val_summary
algo_score.loc[['Random Forest-Categorical Features']]
```

Out[78]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Va (A
Random Forest-Categorical Features	0.290989	0.346424	0.309405	0.391217	0.524161	0.44026	0.418383	0.496303	(

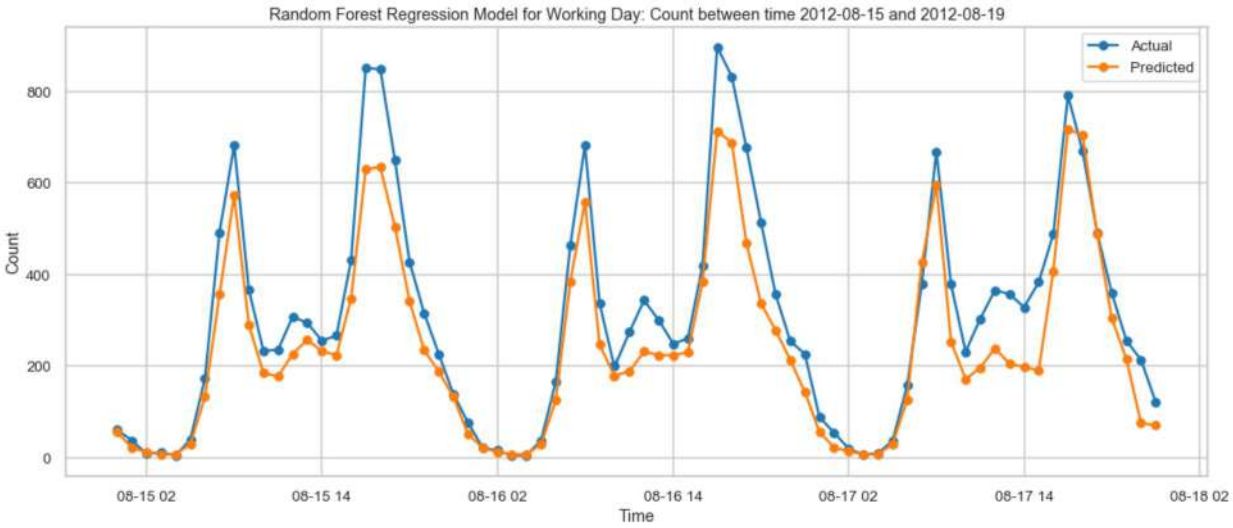
In [79]:
```
algo_score.loc['Random Forest-Categorical Features', 'Training+Test Time (sec)'] = 6.5
cv_time.append(23.9)
```

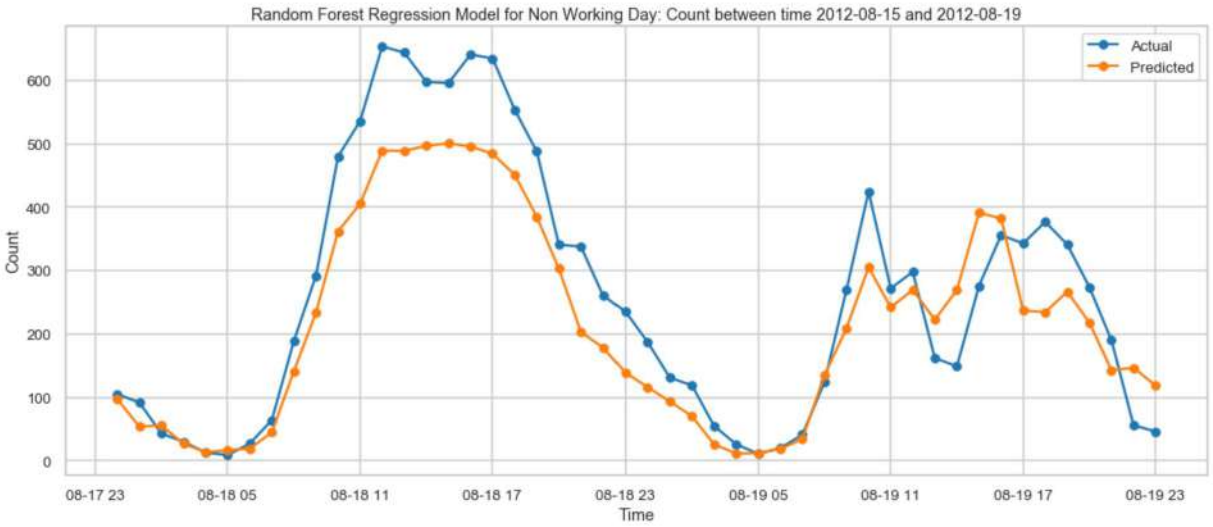
Observations

- Average Test RMSLE = 0.44 which is still a bit higher than 4a Random Forest model
- The average training RMSLE = 0.31 indicates that the model might be a bit overfit

Prediction Plot for Test data in a month (last 5 days of a month)

In [80]:
```
# Random Forest Regression Plot: True vs. Predicted for one month
t_from, t_to = '2012-08-15', '2012-08-19'
ytest_w_predict, ytest_nw_predict = y_predict_summary[1], y_predict_summary[3]
ytest_w_predict = pd.Series(ytest_w_predict, index = ytest_w.index)
ytest_nw_predict = pd.Series(ytest_nw_predict, index = ytest_nw.index)

plot_true_vs_pred(ytest_w, ytest_nw, ytest_w_predict, ytest_nw_predict, 'Random Forest
```



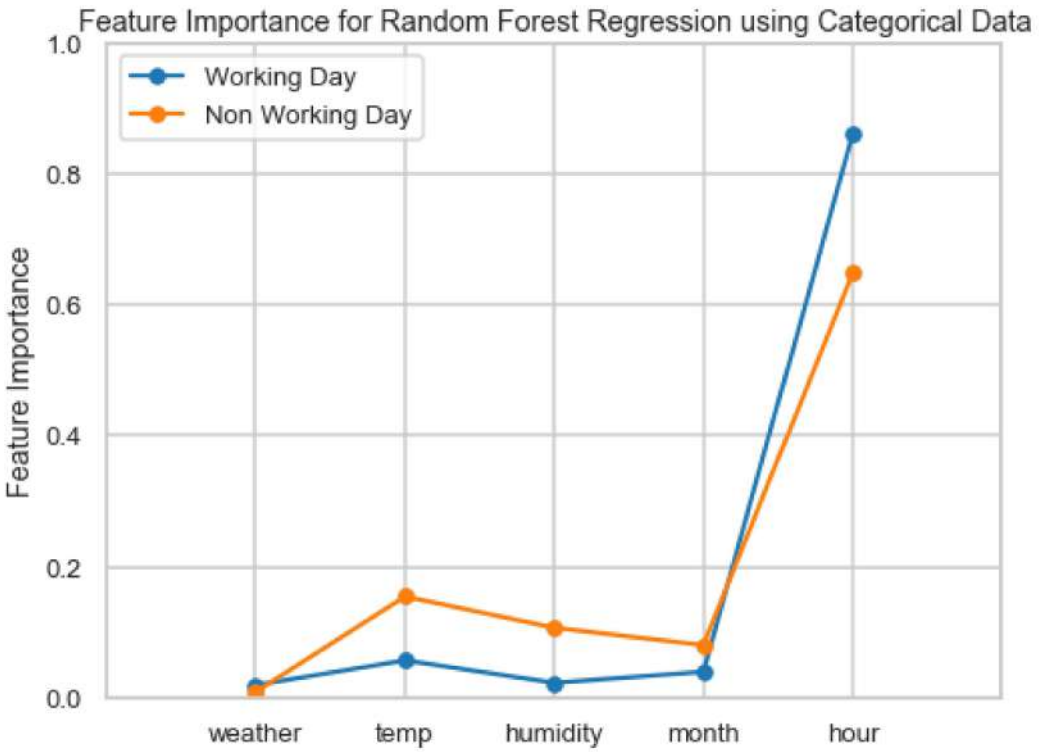


Observations

- Using hour, weather and month in its raw categorical form seem to improve our Model prediction accuracy

Feature Importance

In [81]:
```
# Plotting the Feature Importance
fig = plt.figure(figsize=(8, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(rfc_w.feature_importances_, label='Working Day', marker='.', markersize=15)
axes.plot(rfc_nw.feature_importances_, label='Non Working Day', marker='.', markersize
plt.xticks(range(len(rfc_w.feature_importances_)), X2_w.columns)
axes.set(ylabel='Feature Importance', title='Feature Importance for Random Forest Regr
axes.set(xlim=[-1, len(X2_w.columns)], ylim=[0, 1])
axes.legend()

plt.show()
```



Observations

- The feature importance is similar to the ones in our 4a. model

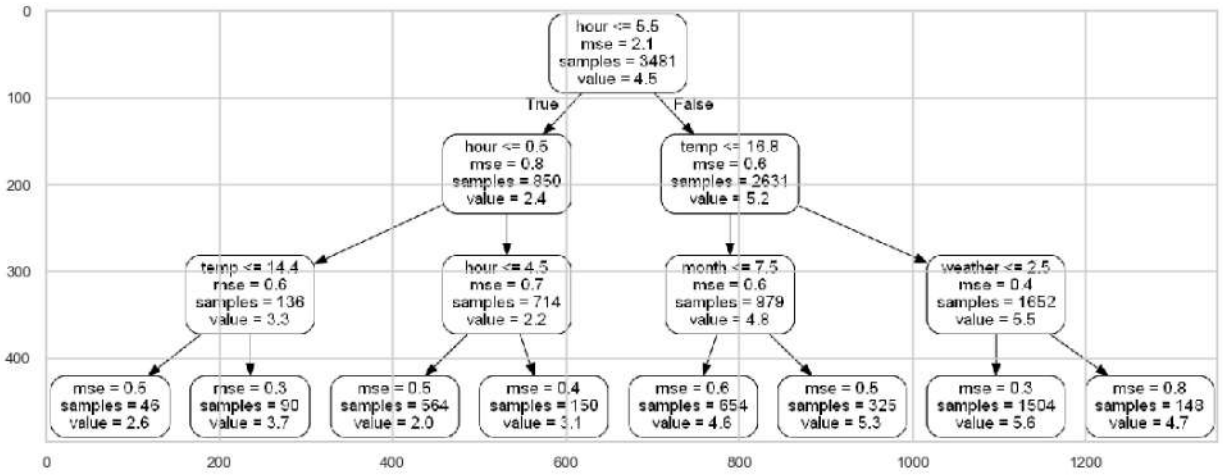
Decision Tree Visualization

Random Forest uses an ensemble of 500 decision trees in our earlier model. Let us visualize one of them and see if it makes sense. The entire tree would have several leaf nodes and would be difficult to analyze. Instead, let us limit the max_depth to 3 and plot the decision tree.

In [82]:
```
# Visualizing a graph limiting the tree to 3 levels to gain some understanding
rf_w_small = RandomForestRegressor(n_estimators=10, max_features = 'auto', max_depth=3
rf_w_small.fit(X2_w, logy2_w)
logy2_w_predict_small = rf_w_small.predict(X2_w)
rmsle2_w_small = rmsle(y2_w, np.expm1(logy2_w_predict_small))
print('RMSLE score for Random Forest Regressor using a small tree (max_depth = 3) for

# Import tools needed for visualization
from sklearn.tree import export_graphviz
import pydot
import matplotlib.image as mpimg

# Pull out one tree from the forest
tree = rf_w_small.estimators_[0]
# Export the image to a dot file
dot_data = export_graphviz(tree, out_file = './Images/tree_rf3.dot', feature_names = X
# Use dot file to create a graph
(graph, ) = pydot.graph_from_dot_file('./Images/tree_rf3.dot')
# Write graph to a png file
graph.write_png('./Images/tree_rf3.png')

img=mpimg.imread('./Images/tree_rf3.png')
plt.figure(figsize=(18, 12))
```

```
imgplot = plt.imshow(img)
plt.show()
```

```
RMSLE score for Random Forest Regressor using a small tree (max_depth = 3) for simple
visualization = 0.644
```



Observations

- Since we limited the above tree to max_depth = 3, we see only 8 leaf nodes. Based on 3 condition checks (at each depth), we arrive at one of these 8 nodes.
- The value indicated at the leaf node corresponds to the predicted value (which is log(1+count))
- The first split is made based on hour feature indicating that it is very important (since that first split at 5.5 hours minimizes the mse) which makes sense
- Note that the number of samples = 3481 at the root node (instead of 5507 which is the total number of observations in X_w). THis is because Random Forest Regression works on a bootstrap sample, and the above bootstrap sample contained just 4868 unique observation

Ensemble Method - Gradient Boost

Two Separate Models + Binary Vector Features via OneHotEncoder

Hyperparameter Tuning

Procedure adopted to tune the parameters

1. Pick a large n_estimator = 3000
2. Tune max_depth, learning_rate, min_samples_leaf, and max_features via grid search.
3. Increase n_estimators even more (5000) and tune learning_rate again holding the other parameters fixed.

In [83]:
```
## Gradient Boost Regression Grid Search to obtain the best parameters.
## Commented it out since it takes a lot of time to run. Using the best parameters obt

from sklearn.ensemble import GradientBoostingRegressor
```

```
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators':[3000], 'learning_rate':[0.01, 0.02, 0.05, 0.1], 'max_de
#param_grid = {'n_estimators':[3000], 'learning_rate':[0.01], 'max_depth':[6], 'max_fe
#param_grid = {'n_estimators':[3000, 5000], 'learning_rate':[0.002, 0.005, 0.01], 'max
#gb_w = GridSearchCV(GradientBoostingRegressor(random_state=42), param_grid, cv=5, sco
#gb_w.fit(X_w, logy_w)
#print('Best parameters for Working Day Gradient Boosting Regression Model: {}'.format

#param_grid = {'n_estimators':[3000, 5000], 'learning_rate':[0.002, 0.005, 0.01], 'max
#gb_nw = GridSearchCV(GradientBoostingRegressor(random_state=42), param_grid, cv=5, sc
#gb_nw.fit(X_nw, logy_nw)
#print('Best parameters for Non Working Day Gradient Boosting Regression Model: {}'.fo
```

Model Fit + Predict

In [84]:
```
# Gradient Boosting Regression

# Best parameters obtained via GridSearchCV above
best_max_depth, best_max_features, best_min_samples_leaf = 6, 1.0, 5
best_n_estimators_w, best_learning_rate_w = 3000, 0.01
best_n_estimators_nw, best_learning_rate_nw = 3000, 0.005
param_summary = ['n_estimators: {}, learning_rate: {}, max_features: {}, min_samples_l
                 'n_estimators: {}, learning_rate: {}, max_features: {}, min_samples_l
print('Best parameters via GridSearchCV for Working Day:     '+param_summary[0])
print('Best parameters via GridSearchCV for Non Working Day: '+param_summary[1])

gb_w = GradientBoostingRegressor(n_estimators = best_n_estimators_w, learning_rate = b
                                 max_depth=best_max_depth, max_features=best_max_featu
gb_nw = GradientBoostingRegressor(n_estimators = best_n_estimators_nw, learning_rate =
                                  max_depth=best_max_depth, max_features=best_max_feat

rmsle_summary, y_predict_summary = model_fit(gb_w, X_w, Xtest_w, y_w, ytest_w, gb_nw,
ypred_test.loc[Xtest.workingday==1,'GB1'], ypred_test.loc[Xtest.workingday==0,'GB1'] =
```

```
Best parameters via GridSearchCV for Working Day:     n_estimators: 3000, learning_ra
te: 0.01, max_features: 1.0, min_samples_leaf: 5, max_depth: 6
Best parameters via GridSearchCV for Non Working Day: n_estimators: 3000, learning_ra
te: 0.005, max_features: 1.0, min_samples_leaf: 5, max_depth: 6
```

In [85]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(gb_w, X_w, y_w, gb_nw, X_nw, y_nw
ypred_train.loc[X.workingday==1,'GB1'], ypred_train.loc[X.workingday==0,'GB1'] = y_pre

algo_score.loc['Gradient Boosting-OneHotEncoding'] = rmsle_summary+rmsle_val_summary+p
algo_score.loc[['Gradient Boosting-OneHotEncoding']]
```

Out[85]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validatio RMSL (No Workin Day
Gradient Boosting-OneHotEncoding	0.272817	0.299252	0.281356	0.407705	0.550037	0.460327	0.425073	0.55916

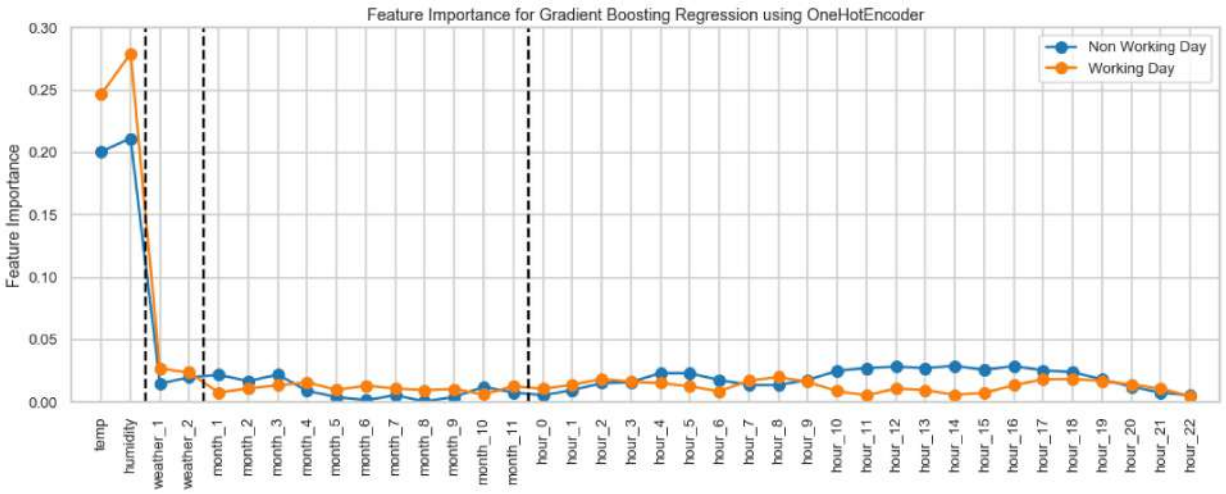
In [86]:
```
algo_score.loc['Gradient Boosting-OneHotEncoding', 'Training+Test Time (sec)'] = 42.9
cv_time.append(210)
```

Observations

- Performs a bit worse compared to the best Random Forest model (4a)
- This again possibly is an overfit model

Feature Importance Plot

In [87]:
```
# Plotting the Feature Importance
fig = plt.figure(figsize=(18, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(gb_nw.feature_importances_, label='Non Working Day', marker='.', markersize=
axes.plot(gb_w.feature_importances_, label='Working Day', marker='.', markersize=20)
plt.xticks(range(len(gb_w.feature_importances_)), X_w.columns, rotation=90)
axes.axvline(2-0.5, c='k', ls='--')
axes.axvline(4-0.5,  c='k', ls='--')
axes.axvline(15-0.5,  c='k', ls='--')
axes.set(ylabel='Feature Importance', title='Feature Importance for Gradient Boosting
axes.set(xlim=[-1, len(X_w.columns)], ylim=[0, 0.3])
axes.legend()

plt.show()
```



Observations

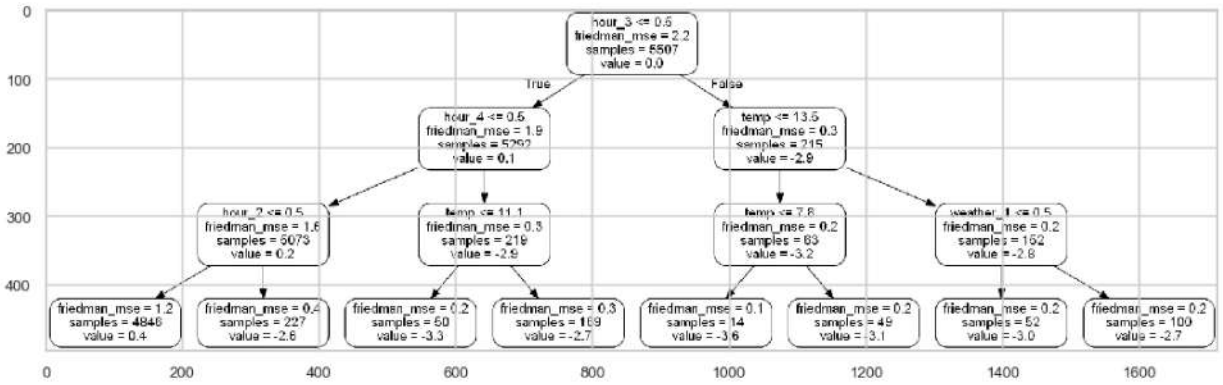
- Higher RMSLE observed compared to Random Forest Regression
- Unlike Random Forest Regression, there isn't a lot of significant difference in the various hours_x features

Decision Tree Visualization

All the individual Decision Tree Estimator used for Gradient Boosting has a depth of 6. Let us use a smaller max_depth (3) to gain some insight

In [88]:
```
# Import tools needed for visualization
from sklearn.tree import export_graphviz
import pydot
import matplotlib.image as mpimg
```

```
gb_w_small = GradientBoostingRegressor(n_estimators = best_n_estimators_w, learning_ra
gb_w_small.fit(X_w, logy_w)

# Pull out one tree from the forest
tree = gb_w_small.estimators_[0][0]
# Export the image to a dot file
dot_data = export_graphviz(tree, out_file = './Images/tree_gb1.dot', feature_names = X
# Use dot file to create a graph
(graph, ) = pydot.graph_from_dot_file('./Images/tree_gb1.dot')
# Write graph to a png file
graph.write_png('./Images/tree_gb1.png')

img=mpimg.imread('./Images/tree_gb1.png')
plt.figure(figsize=(18, 12))
imgplot = plt.imshow(img)
plt.show()
```



Observations

- Each estimator corresponds to 3-level Decision Tree which is applied on the residue obtained till that point
- The very first estimator is the mean of the observed value (mean of the count)
- The above graph corresponds to the first Decision Tree estimator applied on the residue = X_w - mean
- Note that the graph is almost similar to the one obtained before via Random Forest Regression with max_depth = 3. The difference is due to the randomness in the bootstrap sample for Random Forest Method
- Also note that the value in each leaf node corresponds to the estimator on the residue = X_w - mean and hence is different from the ones seen for the Random Forest graph plot

Two Separate Models for Working and Non-working days + Categorical Features

- As with Random Forest approach, let us check if OneHotEncoder provided any benefit

Hyperparameter Tuning

In [89]:
```
## Gradient Boost Regression Grid Search to obtain the best parameters.
## Commented it out since it takes a lot of time to run. Using the best parameters obt
```

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators':[3000], 'learning_rate':[0.005, 0.01, 0.02, 0.05, 0.1],
#param_grid = {'n_estimators':[3000, 5000], 'learning_rate':[0.002, 0.005, 0.01], 'max
#gb2_w = GridSearchCV(GradientBoostingRegressor(random_state=42), param_grid, cv=5, sc
#gb2_w.fit(X2_w, logy2_w)
#print('Best parameters for Working Day Gradient Boosting Regression Model: {}'.format

#gb2_nw = GridSearchCV(GradientBoostingRegressor(random_state=42), param_grid, cv=5, s
#gb2_nw.fit(X2_nw, logy2_nw)
#print('Best parameters for Non Working Day Gradient Boosting Regression Model: {}'.fo
```

Model Fit + Predict

In [90]:
```
# Gradient Boosting Regression

# Best parameters obtained via GridSearchCV above

best_max_depth, best_max_features, best_min_samples_leaf = 4, 1.0, 21
best_n_estimators_w, best_learning_rate_w = 3000, 0.01
best_n_estimators_nw, best_learning_rate_nw = 3000, 0.005
param_summary = ['n_estimators: {}, learning_rate: {}, max_features: {}, min_samples_l
                 'n_estimators: {}, learning_rate: {}, max_features: {}, min_samples_l
print('Best parameters via GridSearchCV for Working Day:     '+param_summary[0])
print('Best parameters via GridSearchCV for Non Working Day: '+param_summary[1])

gb2_w = GradientBoostingRegressor(n_estimators = best_n_estimators_w, learning_rate =
                                  max_depth=best_max_depth, max_features=best_max_featu
gb2_nw = GradientBoostingRegressor(n_estimators = best_n_estimators_nw, learning_rate
                                   max_depth=best_max_depth, max_features=best_max_feat

rmsle_summary, y_predict_summary = model_fit(gb2_w, X2_w, Xtest2_w, y2_w, ytest2_w, gb
ypred_test.loc[Xtest2.workingday==1,'GB2'], ypred_test.loc[Xtest2.workingday==0,'GB2']
```

```
Best parameters via GridSearchCV for Working Day:     n_estimators: 3000, learning_ra
te: 0.01, max_features: 1.0, min_samples_leaf: 21, max_depth: 4
Best parameters via GridSearchCV for Non Working Day: n_estimators: 3000, learning_ra
te: 0.005, max_features: 1.0, min_samples_leaf: 21, max_depth: 4
```

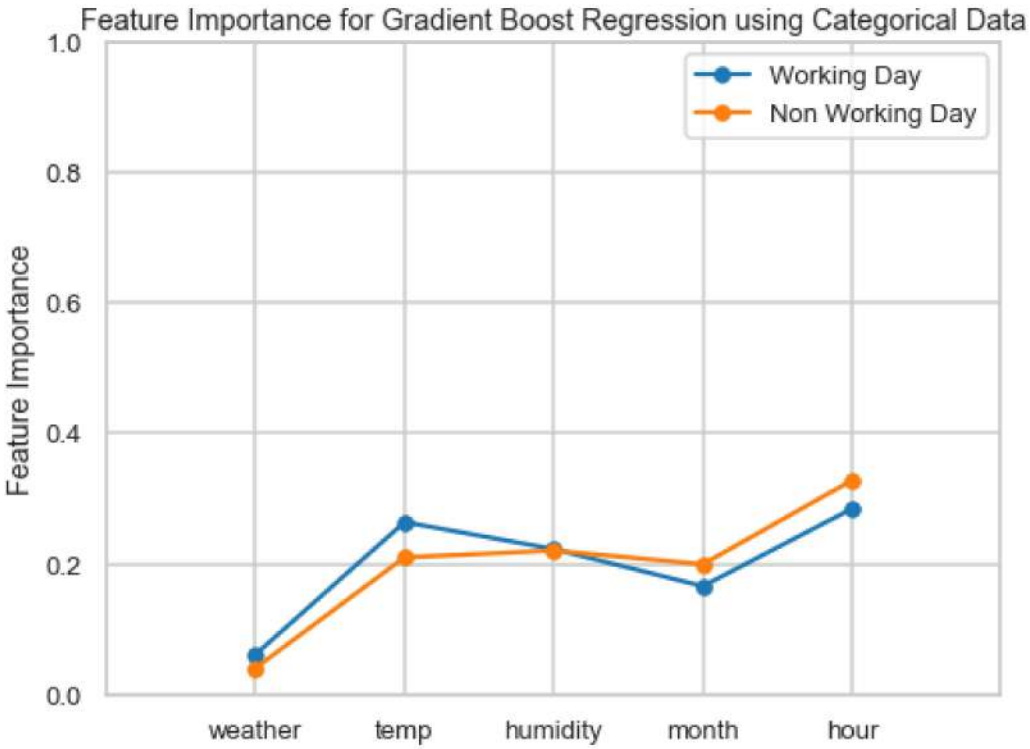
In [91]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(gb2_w, X2_w, y2_w, gb2_nw, X2_nw,
ypred_train.loc[X2.workingday==1,'GB2'], ypred_train.loc[X2.workingday==0,'GB2'] = y_p

algo_score.loc['Gradient Boosting-Categorical Features'] = rmsle_summary+rmsle_val_sum
algo_score.loc[['Gradient Boosting-Categorical Features']]
```

Out[91]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Va (A
Gradient Boosting-Categorical Features	0.312417	0.334973	0.319646	0.387938	0.493577	0.426262	0.404077	0.501422	(

In [92]:
```
algo_score.loc['Gradient Boosting-Categorical Features', 'Training+Test Time (sec)'] =
cv_time.append(42.6)
```

In [93]:
```
# Plotting the Feature Importance
fig = plt.figure(figsize=(8, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(gb2_w.feature_importances_, label='Working Day', marker='.', markersize=15)
axes.plot(gb2_nw.feature_importances_, label='Non Working Day', marker='.', markersize
plt.xticks(range(len(gb2_w.feature_importances_)), X2_w.columns)
axes.set(ylabel='Feature Importance', title='Feature Importance for Gradient Boost Reg
axes.set(xlim=[-1, len(X2_w.columns)], ylim=[0, 1])
axes.legend()

plt.show()
```



Observations

- Performs on par with Random Forest

- Overfit?

Ensemble Method - Adaboost

Hyperparameter Tuning

Procedure adopted to tune the parameters

1. Pick a large n_estimator = 3000 and tune learning_rate for working day model with it
2. Increase n_estimators even more (5000) and tune learning_rate again for both working day and non-working day

In [94]:
```
## AdaBoost Regression Grid Search to obtain the best parameters.
## Commented it out since it takes a lot of time to run. Using the best parameters obt

from sklearn.ensemble import AdaBoostRegressor
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators':[3000], 'learning_rate':[0.0005, 0.001, 0.002, 0.005, 0.
#param_grid = {'n_estimators':[5000, 3000], 'learning_rate':[0.0005, 0.001, 0.002]}
#ab_w = GridSearchCV(AdaBoostRegressor(random_state=42), param_grid, cv=5, scoring=rms
#ab_w.fit(X2_w, logy2_w)
#print('Best parameters for Working Day AdaBoost Regression Model: {}'.format(ab_w.bes

#ab_nw = GridSearchCV(AdaBoostRegressor(random_state=42), param_grid, cv=5, scoring=rm
#ab_nw.fit(X2_nw, logy2_nw)
#print('Best parameters for Non Working Day AdaBoost Regression Model: {}'.format(ab_r
```

Model Fit + Predict

In [95]:
```
# AdaBoost Regression

# Best parameters obtained via GridSearchCV above

best_n_estimators_w, best_learning_rate_w = 5000, 0.001
best_n_estimators_nw, best_learning_rate_nw = 5000, 0.001
param_summary = ['n_estimators: {}, learning_rate: {}'.format(best_n_estimators_w, bes
                 'n_estimators: {}, learning_rate: {}'.format(best_n_estimators_nw, be
print('Best parameters via GridSearchCV for Working Day:     '+param_summary[0])
print('Best parameters via GridSearchCV for Non Working Day: '+param_summary[1])

ab_w = AdaBoostRegressor(n_estimators = best_n_estimators_w, learning_rate = best_lear
ab_nw = AdaBoostRegressor(n_estimators = best_n_estimators_nw, learning_rate = best_le

rmsle_summary, y_predict_summary = model_fit(ab_w, X2_w, Xtest2_w, y2_w, ytest2_w, ab_
ypred_test.loc[Xtest2.workingday==1,'AB'], ypred_test.loc[Xtest2.workingday==0,'AB'] =
```

```
Best parameters via GridSearchCV for Working Day:     n_estimators: 5000, learning_ra
te: 0.001
Best parameters via GridSearchCV for Non Working Day: n_estimators: 5000, learning_ra
te: 0.001
```

In [96]:
```
rmsle_val_summary, y_predict_val_summary = cross_val(ab_w, X2_w, y2_w, ab_nw, X2_nw, y
ypred_train.loc[X2.workingday==1,'AB'], ypred_train.loc[X2.workingday==0,'AB'] = y_pre
```

```
algo_score.loc['AdaBoost-OneHotEncoding'] = rmsle_summary+rmsle_val_summary+param_summ
algo_score.loc[['AdaBoost-OneHotEncoding']]
```

Out[96]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validatio RMSL (No Workin Day
AdaBoost-OneHotEncoding	0.604791	0.558883	0.590809	0.579941	0.651514	0.604868	0.621749	0.598

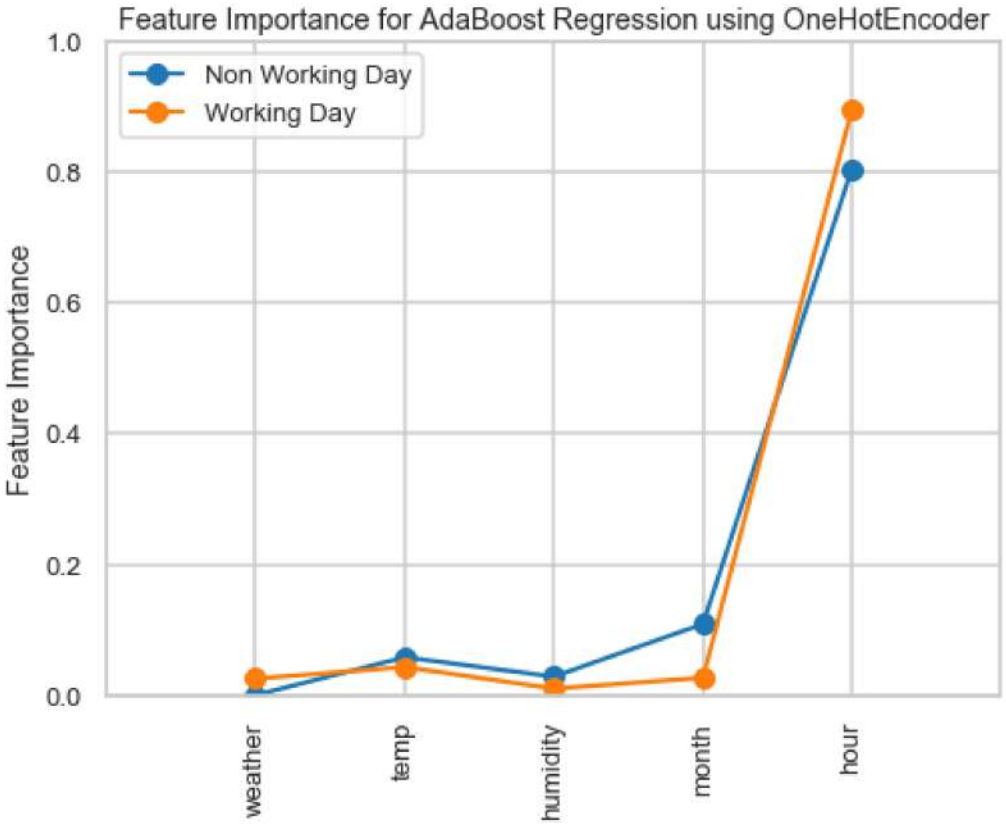
In [97]:
```
algo_score.loc['AdaBoost-OneHotEncoding', 'Training+Test Time (sec)'] = 59.2
cv_time.append(198)
```

Observations

- AdaBoost model has higher RMSLE and doesnt perform as well as the other 2 Ensemble methods (RF and GB)

Feature Importance Plot

In [98]:
```
# Plotting the Feature Importance
fig = plt.figure(figsize=(8, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(ab_nw.feature_importances_, label='Non Working Day', marker='.', markersize=
axes.plot(ab_w.feature_importances_, label='Working Day', marker='.', markersize=20)
plt.xticks(range(len(ab_w.feature_importances_)), X2_w.columns, rotation=90)
axes.set(ylabel='Feature Importance', title='Feature Importance for AdaBoost Regressio
axes.set(xlim=[-1, len(X2_w.columns)], ylim=[0, 1])
axes.legend()

plt.show()
```

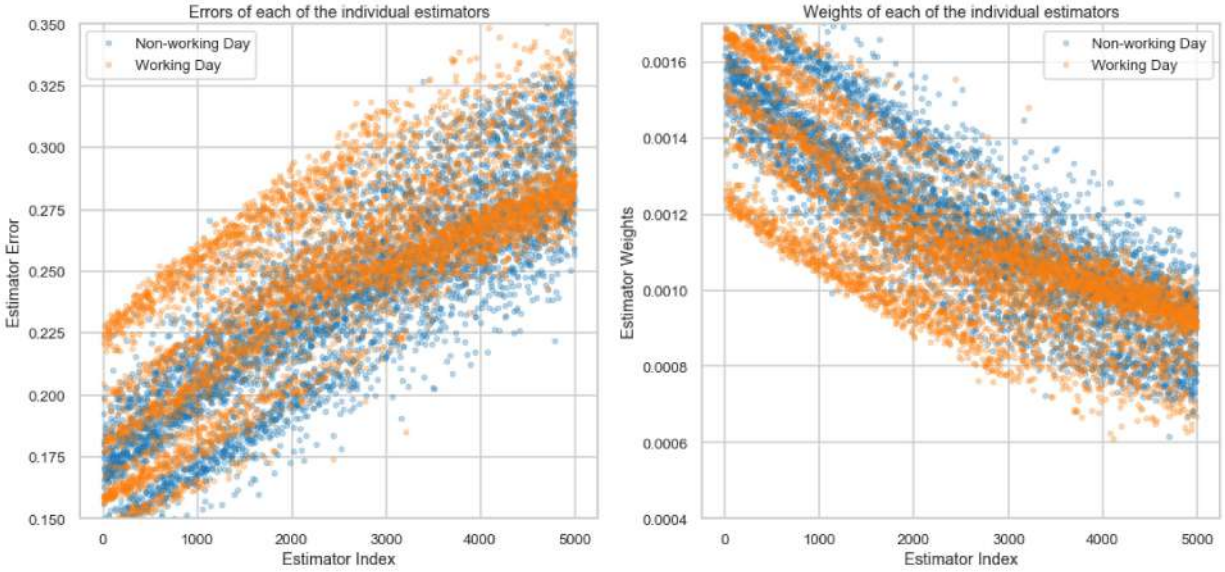


Observations

- Like every other models, AdaBoostRegressor too prioritize the hour features

Plots of Estimator Error and Estimator Weights

In [99]:
```
fig = plt.figure(figsize=(18, 8))

# Estimator Error Plot
axes = fig.add_subplot(1, 2, 1)
axes.plot(ab_nw.estimator_errors_, linestyle='None', marker='.', alpha=0.3, label = 'N
axes.plot(ab_w.estimator_errors_, linestyle='None', marker='.', alpha=0.3, label='Work
axes.set(xlabel='Estimator Index', ylabel='Estimator Error', title='Errors of each of
axes.set(ylim=[0.15, 0.35])
axes.legend()

# Estimator Weight Plot
axes = fig.add_subplot(1, 2, 2)
axes.plot(ab_nw.estimator_weights_, linestyle='None', marker='.', alpha=0.3, label = '
axes.plot(ab_w.estimator_weights_, linestyle='None', marker='.', alpha=0.3, label = 'W
axes.set(xlabel='Estimator Index', ylabel='Estimator Weights', title='Weights of each
axes.set(ylim=[0.0004, 0.0017])
axes.legend()
plt.show()
```



Observations

- From the above plot, it AdaBoost might not be a good model for this problem. Every subsequent estimator seems to result in an increase in the error (and as a result, we give it a lower weight for our final combined estimator model)

Stacking using Linear Regression

Model stacking is an efficient ensemble method in which the predictions, generated by using various machine learning algorithms, are used as inputs in a second-layer learning algorithm. This second-layer algorithm is trained to optimally combine the model predictions to form a new set of predictions. Next few models will apply the stacking concept by using the predictions obtained from above individual models (3.2 to 3.6) as meta-features to build a new ensemble model.

First let us use Linear Regression as the second layer model - this would estimate the weight for each of the individual model predictions by minimizing the least square errors.

Preparing data for Stacking

In [100…
```
X_stack, Xtest_stack = ypred_train.drop('count', axis=1), ypred_test.drop('count', axi
y_stack, ytest_stack = ypred_train['count'], ypred_test['count']
```

Applying Linear Regression Modeling..

In [101…
```
# Stacking using Linear Regressor
param_summary = ['', '', '']

lr_stack = LinearRegression()
rmsle_summary, y_predict_summary = stack_model_fit(lr_stack, X_stack, Xtest_stack, y_s
rmsle_val_summary = ['', '', '']
```

```
algo_score.loc['Stacking-LR'] = rmsle_summary+rmsle_val_summary+param_summary
algo_score.loc[['Stacking-LR']]
```

Out[101]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Vali (Av
Stacking-LR	0.436751	0.499553	0.457331	0.402243	0.48662	0.432355			

In [102…

```
algo_score.loc['Stacking-LR', 'Training+Test Time (sec)'] = sum(cv_time) +0.198
```

Lets take a look at the coefficients obtained for the Linear Regression Model

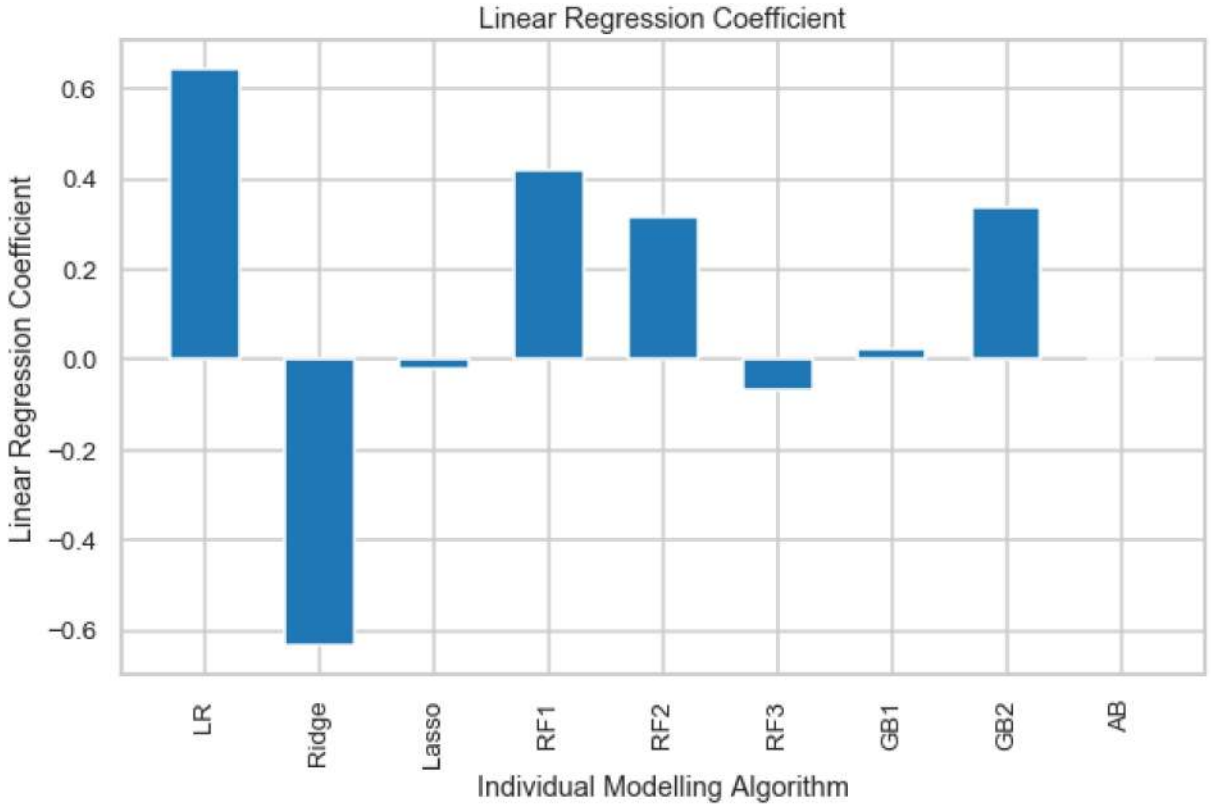
In [103…

```
df_stack_coeff = pd.DataFrame({'Individual Models': X_stack.columns, 'LR Coeff': lr_st
print('Sum of Coeff: ', lr_stack.coef_.sum())
```

```
Sum of Coeff:  1.022788867796151
```

In [104…

```
fig=plt.figure(figsize=(10, 6))
axes=fig.add_subplot(1, 1, 1)
bar_width = 0.6
idx = np.array(range(len(lr_stack.coef_)))
labels = X_stack.columns
plt.bar(height=lr_stack.coef_, x=idx, width=bar_width)
plt.xticks(idx, labels, rotation=90)
plt.xlabel('Individual Modelling Algorithm')
plt.ylabel('Linear Regression Coefficient')
plt.title('Linear Regression Coefficient')
#axes.set_yscale('log')
plt.show()
```



Observations

- Stacking using Linear Regression results in poorer test performance (0.46 RMSLE) compared to few ensemble individual models (eg. Random Forest 3.5.1 with 0.43 RMSLE)
- Sum of the coefficient ~1, which makes sense. We would expect the individual models to have almost a zero bias
- Linear Regression might not work very well since each columns (or features) are highly correlated

Stacking using Random Forest

Hyperparameter Tuning

In [105…

```
## Random Forest Regression Hyperparameter tuning using Grid Search to obtain the best
## Commented it out since it takes a lot of time to run. Using the best parameters obt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators': [500,1000, 2000], 'max_features':['auto', 'sqrt']}
#rf = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scoring=r
#rf.fit(X_stack, np.log1p(y_stack))

#param_grid = {'n_estimators': [1000], 'max_features':['sqrt'], 'min_samples_leaf':[10
#rf = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scoring=r
#rf.fit(X_stack, np.log1p(y_stack))

#param_grid = {'n_estimators': [1000], 'max_features':['sqrt'], 'min_samples_leaf':[10
#rf = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=5, scoring=r
#rf.fit(X_stack, np.log1p(y_stack))
```

```
#print('Best parameters for Random Forest Regression Model: {}'.format(rf.best_params_
```

In [106…

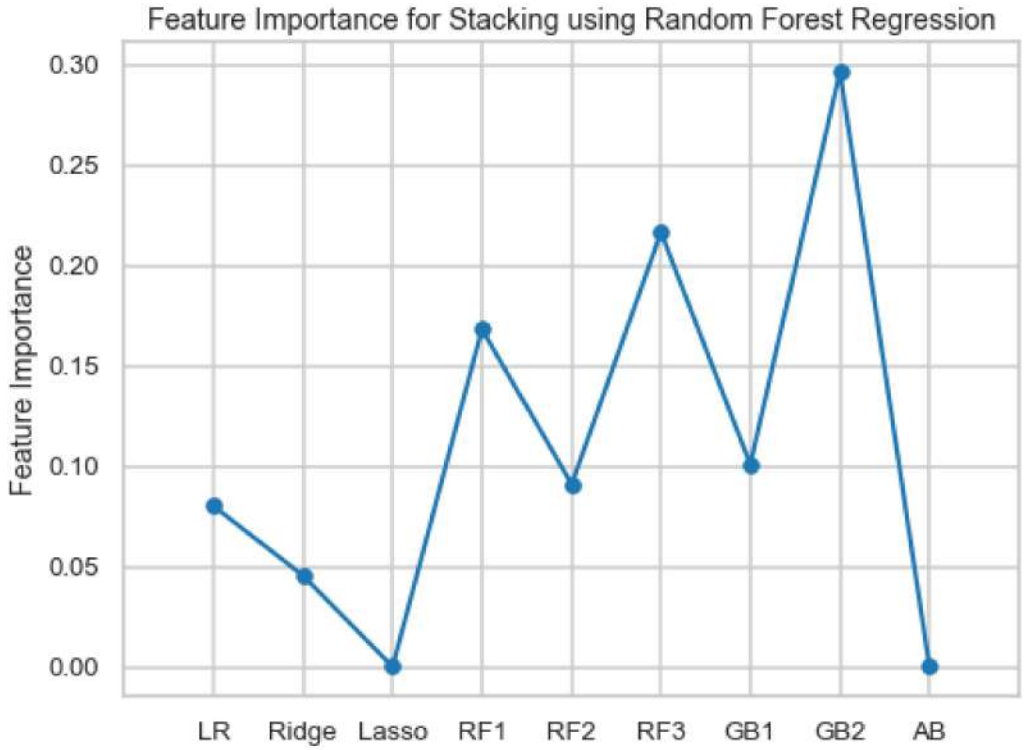
```
best_n_estimators, best_max_features = 1000, 'sqrt'
best_min_samples_leaf, best_max_depth = 100, 5
param_summary = ['n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:
                 'n_estimators: {}, max_features: {}, min_samples_leaf: {}, max_depth:

rf_stack = RandomForestRegressor(n_estimators = best_n_estimators, max_features = best
                                 min_samples_leaf = best_min_samples_leaf, max_depth =

rmsle_summary, y_predict_summary = stack_model_fit(rf_stack, X_stack, Xtest_stack, y_s
rmsle_val_summary = ['', '', '']

algo_score.loc['Stacking-RF'] = rmsle_summary+rmsle_val_summary+param_summary
algo_score.loc[['Stacking-RF']]
```

Out[106]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Vali (Av
Stacking-RF	0.422611	0.472015	0.438669	0.3899	0.494256	0.427714			

In [107…

```
# Plotting the Feature Importance
fig = plt.figure(figsize=(8, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(rf_stack.feature_importances_, marker='.', markersize=15)
plt.xticks(range(len(rf_stack.feature_importances_)), X_stack.columns)
axes.set(ylabel='Feature Importance', title='Feature Importance for Stacking using Ran
axes.set(xlim=[-1, len(X_stack.columns)])

plt.show()
```



In [108…
```
algo_score.loc['Stacking-RF', 'Training+Test Time (sec)'] = sum(cv_time)+9.13
```

Stacking using Gradient Boost

In [109…
```
## Gradient Boost Regression Grid Search to obtain the best parameters.
## Commented it out since it takes a lot of time to run. Using the best parameters obt

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV

#param_grid = {'n_estimators':[3000], 'learning_rate':[0.001, 0.005], 'max_depth':[4,
#param_grid = {'n_estimators':[3000], 'learning_rate':[0.001], 'max_depth':[4], 'max_f
#param_grid = {'n_estimators':[3000], 'learning_rate':[0.001], 'max_depth':[4], 'max_f
#param_grid = {'n_estimators':[3000], 'learning_rate':[0.001, 0.005, 0.01], 'max_depth
#gb = GridSearchCV(GradientBoostingRegressor(random_state=42), param_grid, cv=5, scori
#gb.fit(X_stack, np.log1p(y_stack))
#print('Best parameters for Working Day Gradient Boosting Regression Model: {}'.format
```

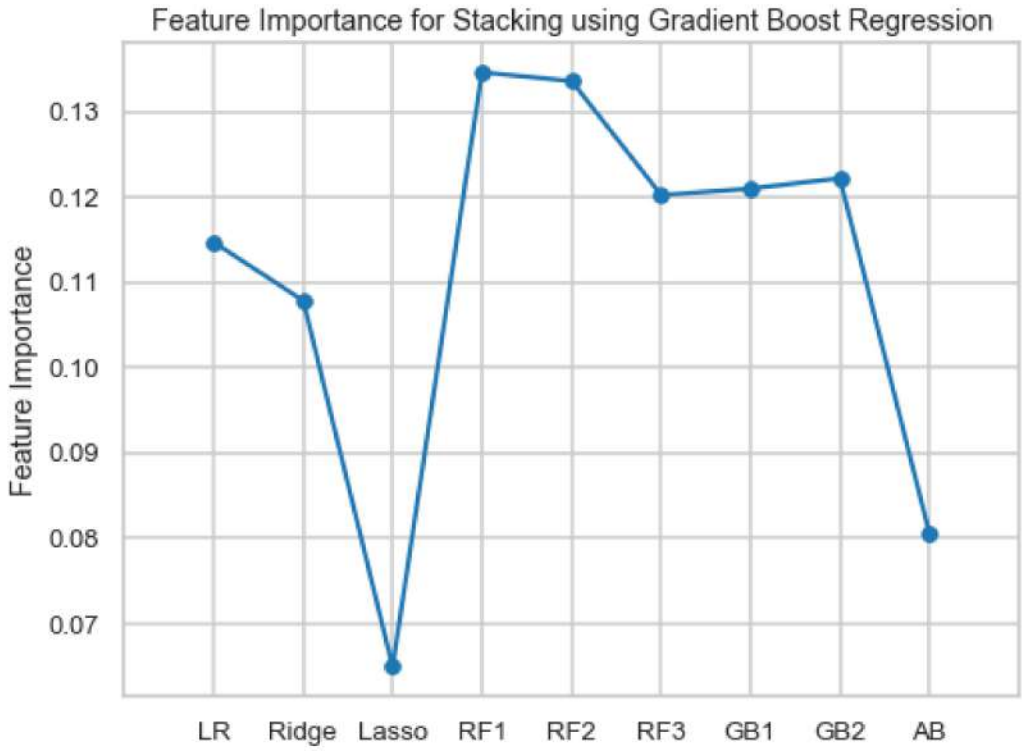
In [110…
```
best_max_depth, best_max_features, best_min_samples_leaf = 4, 0.3, 100
best_n_estimators, best_learning_rate = 3000, 0.01

param_summary = ['n_estimators: {}, learning_rate: {}, max_features: {}, min_samples_l
                 'n_estimators: {}, learning_rate: {}, max_features: {}, min_samples_l

gb_stack = GradientBoostingRegressor(n_estimators = best_n_estimators, learning_rate =
                                     max_depth=best_max_depth, max_features=best_max_featu

rmsle_summary, y_predict_summary = stack_model_fit(gb_stack, X_stack, Xtest_stack, y_s
rmsle_val_summary = ['', '', '']

algo_score.loc['Stacking-GB'] = rmsle_summary+rmsle_val_summary+param_summary
algo_score.loc[['Stacking-GB']]
```

Out[110]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validation RMSLE (Non Working Day)	Vali (Av
Stacking-GB	0.408437	0.458788	0.424835	0.393135	0.494927	0.429937			

In [111…

```
# Plotting the Feature Importance
fig = plt.figure(figsize=(8, 6))
axes = fig.add_subplot(1, 1, 1)
axes.plot(gb_stack.feature_importances_, marker='.', markersize=15)
plt.xticks(range(len(gb_stack.feature_importances_)), X_stack.columns)
axes.set(ylabel='Feature Importance', title='Feature Importance for Stacking using Gra
axes.set(xlim=[-1, len(X_stack.columns)])

plt.show()
```



In [112…

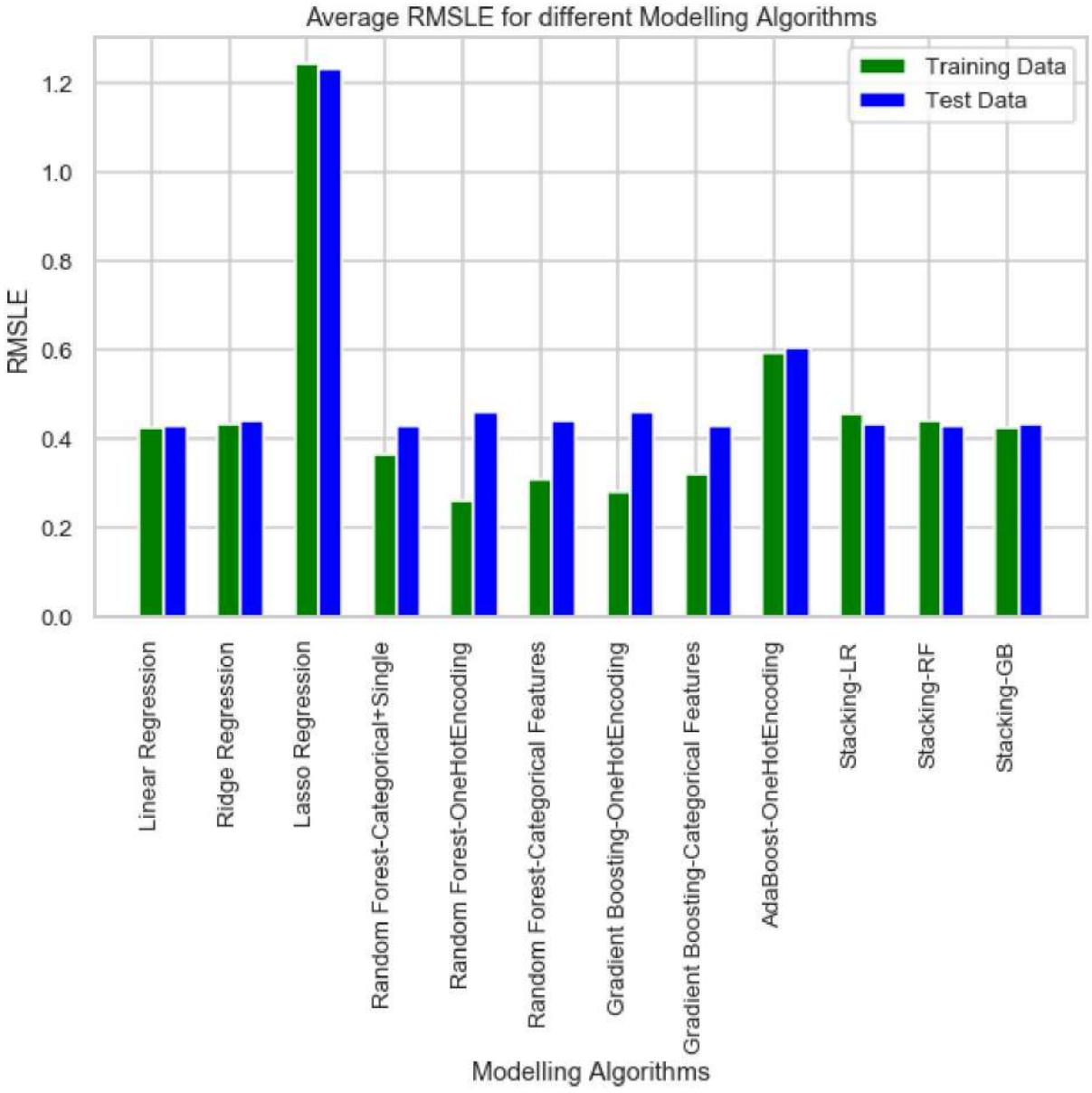
```
algo_score.loc['Stacking-GB', 'Training+Test Time (sec)'] = sum(cv_time)+8.12
```

SUMMARY AND CONCLUSIONS

RMSLE

In [113…

```
fig=plt.figure(figsize=(10, 6))
axes=fig.add_subplot(1, 1, 1)
bar_width = 0.3
idx = np.array(range(algo_score.shape[0]))
labels = algo_score.index
plt.bar(data=algo_score, height='Train RMSLE (Average)', x=idx, color='g', width=bar_w
plt.bar(data=algo_score, height='Test RMSLE (Average)', x=idx+bar_width, color='b', wi
plt.xticks(idx, labels, rotation=90)
plt.xlabel('Modelling Algorithms')
plt.ylabel('RMSLE')
plt.title('Average RMSLE for different Modelling Algorithms')
plt.legend()
plt.show()
```



In [114…

```
# Detailed split of RMSLE
algo_score
```

Out[114]:

Modelling Algo	Train RMSLE (Working Day)	Train RMSLE (Non Working Day)	Train RMSLE (Average)	Test RMSLE (Working Day)	Test RMSLE (Non Working Day)	Test RMSLE (Average)	Validation RMSLE (Working Day)	Validat RM (N Work D
Linear Regression	0.418155	0.432817	0.422797	0.390881	0.495181	0.428666	0.430658	0.474
Ridge Regression	0.423189	0.448141	0.431152	0.394230	0.516847	0.439148	0.438216	0.492
Lasso Regression	1.313353	1.065252	1.241062	1.285346	1.117959	1.231794	1.31459	1.07
Random Forest-Categorical+Single	0.353823	0.387612	0.364732	0.393007	0.489250	0.427676	0.425339	0.482
Random Forest-OneHotEncoding	0.181652	0.378901	0.259992	0.409380	0.541029	0.457731	0.437254	0.53
Random Forest-Categorical Features	0.290989	0.346424	0.309405	0.391217	0.524161	0.440260	0.418383	0.496
Gradient Boosting-OneHotEncoding	0.272817	0.299252	0.281356	0.407705	0.550037	0.460327	0.425073	0.559
Gradient Boosting-Categorical Features	0.312417	0.334973	0.319646	0.387938	0.493577	0.426262	0.404077	0.501
AdaBoost-OneHotEncoding	0.604791	0.558883	0.590809	0.579941	0.651514	0.604868	0.621749	0.5
Stacking-LR	0.436751	0.499553	0.457331	0.402243	0.486620	0.432355		
Stacking-RF	0.422611	0.472015	0.438669	0.389900	0.494256	0.427714		
Stacking-GB	0.408437	0.458788	0.424835	0.393135	0.494927	0.429937		

Based on the above result, let us use Single Model Random Forest Regressor Model (with categorical feature) based on the below couple of reasons

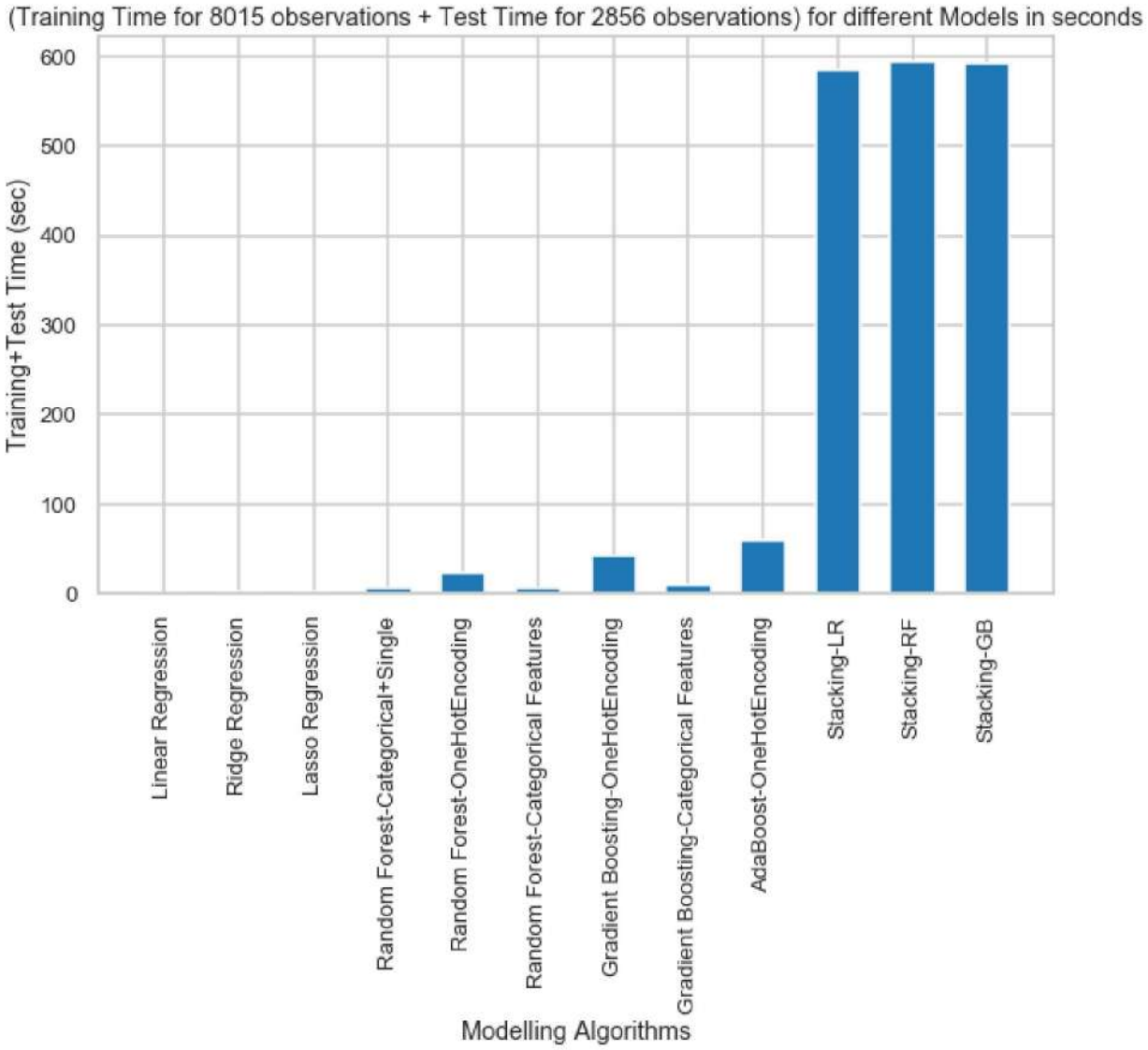
- One of the lowest Average Test RMSLE
- Single model which works for both working and non-working days
- Least Overfit (Train Average RMSLE is closer to Test Average RMSLE)
- Lesser to train due to lesser number of features

Train/Test Time

Plot of the time taken by each of the algorithm to train the model over 8015 observations and obtain predictions on the 2856 test data

In [115…

```
fig=plt.figure(figsize=(10, 6))
axes=fig.add_subplot(1, 1, 1)
bar_width = 0.6
idx = np.array(range(algo_score.shape[0]))
labels = algo_score.index
plt.bar(data=algo_score, height='Training+Test Time (sec)', x=idx, width=bar_width)
plt.xticks(idx, labels, rotation=90)
plt.xlabel('Modelling Algorithms')
plt.ylabel('Training+Test Time (sec)')
plt.title('(Training Time for 8015 observations + Test Time for 2856 observations) for
#axes.set_yscale('log')
plt.show()
```



Our chosen algorithm, 'Random Forest-Categorical+Single' has a reasonably low train+test time

Kaggle Submission

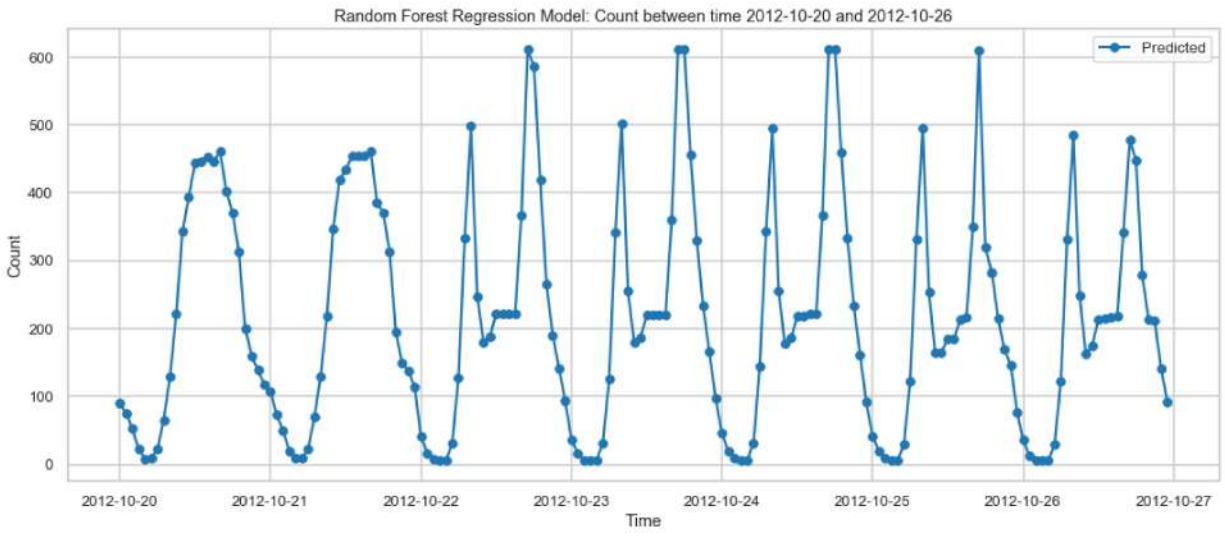
Now that we have picked which model to go with, let us train our model with the entire dataset provided.

In [116…

```
Xtrain = mydata_without_outliers.drop('count', axis=1)
ytrain = mydata_without_outliers['count']
logytrain = np.log1p(ytrain)

# Best parameters obtained via GridSearchCV above
best_n_estimators, best_max_features = 500, 'auto'
best_min_samples_leaf, best_max_depth = 7, 10

rf_main = RandomForestRegressor(n_estimators = best_n_estimators, max_features = best_
                                min_samples_leaf = best_min_samples_leaf, max_depth =
rf_main.fit(Xtrain, logytrain)
logytest_predict = rf_main.predict(testdata)
ytest_predict = np.expm1(logytest_predict)

submission = pd.DataFrame({'datetime':testdata.index, 'count':ytest_predict})
submission.to_csv('./Data/bikeSharing_submission.csv', index=False)

# This will yield a Test Score = 0.49!
```

In [117…

```
algo = 'Random Forest Regression'
t_from, t_to = '2012-10-20', '2012-10-26'
ytest_predict = pd.Series(ytest_predict, index = testdata.index)
fig = plt.figure(figsize=(18, 16))
# Working day plot
axes = fig.add_subplot(2, 1, 1)
axes.plot(ytest_predict[t_from:t_to], label='Predicted', marker='.', markersize=15)
axes.set(xlabel='Time', ylabel='Count', title='{0} Model: Count between time {1} and {
axes.legend()

plt.show()
```



Summary

In this report, we train a model to predict the number of bike rentals at any hour of the year given the weather conditions. The data set was obtained from the Capital Bikeshare program in

Washington, D.C. which contained the historical bike usage pattern with weather data spanning two years.

First we do Exploratory Data Analysis on the data set. We look for missing data values (none were found) and outliers and appropriately modify them. We also perform correlation analysis to extract out the important and relevant feature set and later perform feature engineering to modify few existing columns and drop out irrelavant ones.

We then look at several popular individual models from simple ones like Linear Regressor and Regularization Models (Ridge and Lasso) to more complicated ensemble ones like Random Forest, Gradient Boost and Adaboost. Additionally, few options for model formulation were tried - 1. A single unified model for working and non-working days, 2. Two separate models for working and non-working days, 3. Using OneHotEncoding to get Binary Vector representation of Categorical Features and 4. Using Categorical Features as provided. Finally, we also tried stacking algorithms where the predictions from the level 1 individual models were used as meta-features into a second level model (Linear Regressor, Random Forest and Gradient Boost) to further enhance the predicting capabilities.

The labeled data set provided comprised of the first 19 days of each month while the Kaggle Test Data comprised of rental information from 20th to the end of the month. Hyperparameters were tuned using GridSearchCV cross validataion using 5 folds on part of the provided training data set (first 14 days of each month). The remaining data (15th to 19th of each month) were used as hold out set to test our model performance. Of all the methods and models, we found Random Forest Ensemble method using a Single Model and Categorical Feature set an ideal choice with train and test scores of 0.36 and 0.427, respectively. The training and test time for the chosen model are very reasonable (~23 seconds for total train+test observation size = 10871). The chosen model yields a score of 0.49 on Kaggle Test Data.

Data Exploration Conclusions

In this project, we explored several types of informations that influence bike rental count. Below is a quick summary of exploratory data analysis

- **Working or Non-working Day** We see 2 rental patterns across the day in bike rentals count - first for a `Working Day` where the rental count high at peak office hours (8am and 5pm) and the second for a `Non-working day` where rental count is more or less uniform across the day with a peak at around noon.
- **Hour of the day:** Bike rental count is mostly correlated with the time of the day. As indicated above, the count reaches a high point during peak hours on a working day and is mostly uniform during the day on a non-working day
- **Casual and Registered Users:** While most casual users are likely to be tourists whose rental count is high during non working days, most registered users are most likely city natives whose rental count is high during working days
- **Temperature:** People generally prefer to bike at moderate to high temperatures. We see highest rental counts between 32 to 36 degree celcius

- **Season:** We see highest number bike rentals in Fall (July to September) and Summer (April to June) Seasons and the lowest in Spring (January to March) season
- **Weather:** As one would expect, we see highest number of bike rentals on a clear day and the lowest on a snowy or rainy day
- **Humidity:** With increasing humidity, we see decrease in the number of bike rental count.

Modeling Conclusions

We used 6 Regression Models to predict the bike rental count at any hour of the day - Linear Regression, Ridge, Lasso, Random Forest, Gradient Boost and Adaboost. Using the predictions made by these level 1 individual models as features, we trained 3 level 2 stacking algorithms (Linear Regression, Random Forest and Gradient Boost) to make more refined predictions. Below is a summary of the model performances

- Of all the models, we found a simple Random Forest Model providing the best/lowest RMSLE score.
- Stacking individual models didnt provide any improvement over the best individual model
- Having separate models for Working days and Non-working days didnt provide any improvment in prediction accuracy

Limitations and Scope for Future Work

Below are few limitations in this analysis and ideas to improve model prediction accuracy

- Since casual + registered = total count, we just predicted the total bike rental count by ignoring the casual and registered user information. Another (possibly better) method would be to train separate models for casual and registered users and add the two
- One limitation in the provided training data set is that it lacked data with extreme weather condition data (weather = 4). Hence we had to modify it to weather = 3
- Windspeed wasnt used due to very low correlation. This might have been due to several instances where windspeed = 0. One possible method could be to first estimate those windspeed and then use it as a feature to estimate count.