Data Cleansing, Exploratory Data Analysis, & Using a Web API

Paul R Phillips

December 20, 2019

Contents

1 Description

In this project I will cover how to connect to the Kaggle API from a Linux terminal, illustrate how to perform data cleansing tasks through utilization of python's powerful pandas tool, as well as use various avaiable tools for visual and statistical exploratory data analysis. One important note to make here is that there is no 'cookie cutter' approach to data cleansing. The procedure is somewhat of an art with this respect. The key principal with data cleansing is to answer the following:

1. What insights are you trying to gain with your obtained information?
2. Are the current column data formats appropriate in regards to insights needed?
3. Do the values of the columns offer relevance to the context of the data's story?

I will briefly show you how to answer these questions through data cleansing in this demonstration

2 Story behind the Bike Sharing Data

London, as many countries in Europe and the UK, have provided a public form of transportation through bike rentals. Each country has its own bike rental organization, but the rental workflows are the same. An individual creates an account at a bike rental kiosk, pays for a certain range of days which they wish to rent bikes, and receives a slip which allows them to grab/return rental bikes from the various hubs located across a given city. The hubs are typically found at major street corners. The purpose of the provided Kaggle data set 'Bike Sharing' is to try to predict future bike shares based on provided information collected.

2.1 Bike Sharing Metadata

Below is the list of parameters within the data set:

- **timestamp** - timestamp field for grouping the data
- **cnt** - the count of a new bike shares
- **t1** - real temperature in C
- **t2** - temperature in C 'feels like'
- **hum** - humidity in percentage
- **wind_speed** - wind speed in km/h
- **weather_code** - category of weather
- **is_holiday** - boolean field - 1 holiday / 0 non holiday
- **is_weekend** - boolean field - 1 if the day is weekend
- **season** - category of field meteorological seasons: 0-spring; 1-summer; 2-fall; 3-winter

2.2 Weather Code Category Description

Below are more provided details about weather code:

- *1* - Clear; mostly clear but have some values with haze/fog/patches of fog/fog in vincinity
- *2* - Scattered clouds / few clouds
- *3* - Broken clouds / few clouds
- *4* - Cloudy

- *7* - Rain/ light Rain shower/ Light rain
- *10* - Rain with thunderstorm
- *26* - Snowfall
- *94* - Freezing fog

Note: Here you might already start to ask, does 2-4 really need to be separate categories? Most people are willing to be outside regardless of what variation of cloud formation a city is experiencing for a given day. These conditions will be condensed into the following categories:

- **good** - perfect or close to perfect riding weather
- **fair** - reasonable riding weather
- **rainy** - category 7 weather code (same as above)
- **unpleasant** - any weather condition that would be considered either unsafe, or hazardous to ride in

More details will be provided later on how this is done.

2.3 Setting Up Ubuntu Desktop for Kaggle API from Terminal

Below is the code for setting up Kaggle API and downloading file from Kaggle

Note: Prior to this code being executed, create a Kaggle account, go to 'My Account', and click 'Create API Token'.

Run the following in your command line

```
# Kaggle installation
pip install kaggle

# Move downloaded API token into appropriate directory
mv ~/Downloads/kaggle.json ~/.kaggle/

# For security, run:
chmod 600 /home/mentum/.kaggle/kaggle.json
```

3 Data Import using Command Line and Cleansing using Python

Note: On my machine I use Anaconda3 and Spyder3 for Python

Here I will provide code for importing data from Kaggle using the command line. In Python, the data will be imported locally in proper indexing structure, as well as how to perform some data cleansing and exploratory data analysis on the fly:

3.1 Import Data from Web API

```
# Download file:
kaggle datasets download hmavrodiev/london-bike-sharing-dataset

# Install unzip (if you don't have it already)
sudo apt-get install unzip
```

```
# Unzip bikesharing file and move into desired path
sudo unzip london-bike-sharing-dataset.zip -d ~/python_projects/bikesharing/

# Delete zip file
rm london-bike-sharing-dataset.zip
```

Time to begin our work in Python!

```
# Import needed packages
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import datetime
from pandas.api.types import CategoricalDtype

# Link to csv file location
location = '/home/mentum/python_projects/bikesharing/london_merged.csv'

# Pulling file
# Note: For time series EDA the timestamp will be parsed and used as index
bikesharing = pd.read_csv(location,
                          delimiter=',',
                          parse_dates=True,
                          index_col='timestamp')

# Inspect raw set:
print(bikesharing.info())
```

Below is the output of the print statement:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 17414 entries, 2015-01-04 00:00:00 to 2017-01-03 23:00:00
Data columns (total 9 columns):
cnt             17414 non-null int64
t1              17414 non-null float64
t2              17414 non-null float64
hum             17414 non-null float64
wind_speed      17414 non-null float64
weather_code    17414 non-null float64
is_holiday      17414 non-null float64
is_weekend      17414 non-null float64
season          17414 non-null float64
dtypes: float64(8), int64(1)
memory usage: 1.3 MB
None
```

Now Let's check for missing values:

```
# Check for na's
print(bikesharing.isnull().sum())
```

Output for print statement:

```
cnt             0
t1              0
```

```
t2               0
hum              0
wind_speed       0
weather_code     0
is_holiday       0
is_weekend       0
season           0
dtype: int64
```

3.2 Converting float64 values to bool

Great no missing data! You may have noticed that **is_holiday** & **is_weekend** are currently stored as float64 objects. If we recall the description mentioned these were intended to be boolean. Let's correct that quick:

```
# Convert is_holiday & is_weekend to boolean
convert_list = ['is_holiday', 'is_weekend']

for var in convert_list:
    bikesharing[var] = bikesharing[var].astype(bool)

# Validate correction:
print(bikesharing.info())
```

Output for print statement:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 17414 entries, 2015-01-04 00:00:00 to 2017-01-03 23:00:00
Data columns (total 9 columns):
cnt             17414 non-null int64
t1              17414 non-null float64
t2              17414 non-null float64
hum             17414 non-null float64
wind_speed      17414 non-null float64
weather_code    17414 non-null float64
is_holiday      17414 non-null bool
is_weekend      17414 non-null bool
season          17414 non-null float64
dtypes: bool(2), float64(6), int64(1)
memory usage: 1.1 MB
None
```

Now that these variables are in proper formatting I wonder how many observations there actually are for weekend and holiday rides. Because we took the time to cleanse the columns we can do this quickly with four print statements!

```
# Explore is_holiday & is_weekend
holiday_ride = bikesharing.is_holiday.sum()
weekend_ride = bikesharing.is_weekend.sum()

print('Number of rows for holiday ride: {}'.format(holiday_ride))
print('Proportion of holiday rides: {}'.format(holiday_ride/len(bikesharing)))
print('Number of rows for weekend ride: {}'.format(weekend_ride))
print('Proportion of weekend rides: {}'.format(weekend_ride/len(bikesharing)))
```

Output of print statements:

```
Number of rows for holiday ride: 384
Proportion of holiday rides: 0.02205122315378431
Number of rows for weekend ride: 4970
Proportion of weekend rides: 0.28540254967267714
```

It looks like holiday rides only makes up 2.2% of the data and weekend rides make up 28.5%! This makes sense most of the data will consist of non-holiday business days. At this surface this may not seem important, but if we wanted to use machine learning our algorithms would especially deem holiday rides as obsolete due to the imbalanced variable. That's risky business as we don't yet understand enough about this data set. Remember this data set was collected with the intention of predicting future bike shares so it's likely there are bike share spikes on holidays and weekends. We will do data exploration on this in the next section.

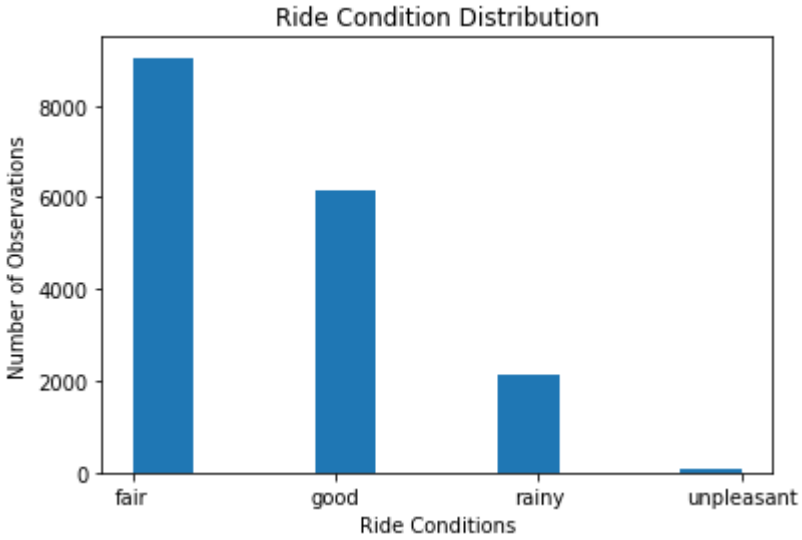
3.3 Mapping Existing Categorical Values to New Values

So far, so good! As noted in the description we now need to handle the **weather_code** variable. Due to the amount, and inconsistency in categorical level values there is interpretability lost with this variable's current state. The code below will condense the existing categorical levels into easier-to-understand levels. We will also create a new categorical variable called **ride_condition**:

```
# Map old categorical levels to new levels
mapping = {1:'good',
           2:'fair',
           3:'fair',
           4:'fair',
           7:'rainy',
           10:'unpleasant',
           26:'unpleasant',
           94:'unpleasant'
           }

# Create new variable 'ride_condition'
bikesharing['ride_condition'] = bikesharing.weather_code.map(mapping)

# Convert to categorical type for plotting:
plot_bikesharing_cats = bikesharing['ride_condition'].astype('category')

# Define categorical levels
cats = ['good', 'fair', 'rainy', 'unpleasant']

# Define categorical levels, and specify ordered parameter to be true
cat_type = CategoricalDtype(cats,
                            ordered=True)

# Create new categorical variable called 'ride_condition'
bikesharing['ride_condition'] = bikesharing['ride_condition'].astype(cat_type)

# Inspect counts of ride_condition:
print(bikesharing.ride_condition.head())
print(bikesharing.ride_condition.value_counts())

# Visualize distribution of observations across weather conditions
plt.hist(plot_bikesharing_cats)
plt.show()
```

Output of print statements:

```
timestamp
2015-01-04 00:00:00    fair
2015-01-04 01:00:00    good
2015-01-04 02:00:00    good
2015-01-04 03:00:00    good
2015-01-04 04:00:00    good
Name: ride_condition, dtype: category
Categories (4, object): [good < fair < rainy < unpleasant]

fair          9049
good          6150
rainy         2141
unpleasant      74
Name: ride_condition, dtype: int64
```



There's a couple notes to make here. You may have noticed a special unordered version of this variable was created. This is because it is computationally expensive to visualize sorted values. Since the weather codes' associated values progressed in descending order, we may want to inherit this trait with our new catgorical variable for later analytical work, which is why the variable in the original data set is defined that way. You may have also noticed that the dominant categorical levels are *good* & *fair*. This makes sense because people are most likely to ride with these conditions present. It's also worth mentioning that combining the original levels *10*, *26*, & *94* only represents 74 observations. Overall it seems our conversion is good representation of the original data.

3.4 Converting Celsius to Fahrenheit using Lambda Functions

Now growing up in the US, I have a hard time relating to temperatures displayed in Celcius. We should be sensitive to this as our audience may be the same way. I already Google'd the formula from converting Celcius temperatures to Fahrenheit. Let me show you how to make this conversion:

```
# Convert Celsius to Fahrenheit
celsius_vars = ['t1', 't2']
fahrenheit_vars = ['t1_F', 't2_F']

for var in celsius_vars:
    for varF in fahrenheit_vars:
        bikesharing[varF] = bikesharing[var].apply(lambda x: (x * 9/5) + 32)


print(bikesharing[['t1', 't2', 't1_F', 't2_F']].head())
```

Output of print statement:

```
                     t1   t2  t1_F  t2_F
timestamp
2015-01-04 00:00:00  3.0  2.0  35.6  35.6
2015-01-04 01:00:00  3.0  2.5  36.5  36.5
2015-01-04 02:00:00  2.5  2.5  36.5  36.5
2015-01-04 03:00:00  2.0  2.0  35.6  35.6
2015-01-04 04:00:00  2.0  0.0  32.0  32.0
```

Fantastic! I can understand that it was under 40 degress F for the January data displayed in 2015. Let's wrap up our initial data cleansing.

3.5 Wrapping Up Initial Data Cleansing

I'll note here that I am calling this the initial data cleansing because these are the actions needed for statistical and exploratory data analysis. Machine learning may require further cleansing. This will be discovered during our EDA section.

Let's convert **wind_speed** to miles per hour:

```
# Convert Km/Hr to M/Hr and store in new variable
bikesharing['wind_speed_mph'] = bikesharing['wind_speed'].apply(lambda x: x/1.609)

# Validate results
print(bikesharing[['wind_speed','wind_speed_mph']].head())
```

Output of print statement:

```
                     wind_speed  wind_speed_mph
timestamp
2015-01-04 00:00:00         6.0        3.729024
2015-01-04 01:00:00         5.0        3.107520
2015-01-04 02:00:00         0.0        0.000000
2015-01-04 03:00:00         0.0        0.000000
2015-01-04 04:00:00         6.5        4.039776
```

Great following the same principal for converting measures of temperature we worth able to quickly convert wind speed measures! You may have noticed that I kept the original windspeed column as is. Just understand that anytime a measurement is converted there is precision lost. This can impact your model's predictive

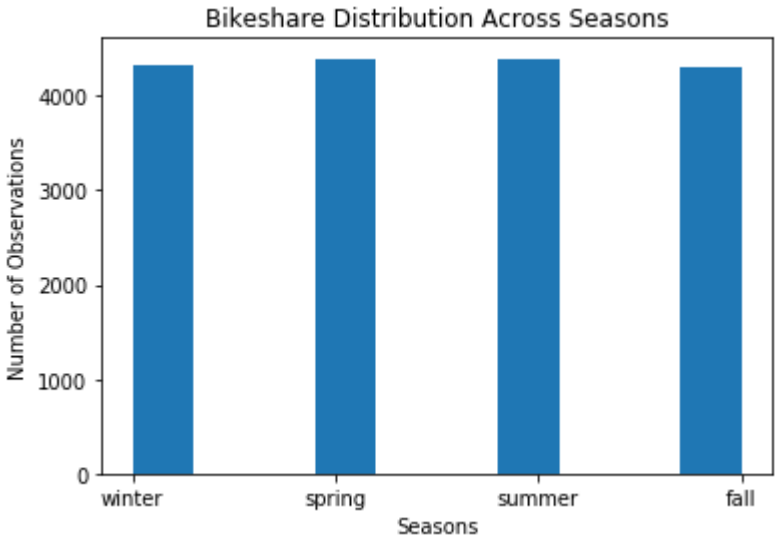
power. I would recommend creating a new variable out of the converted values for EDA, and use the original measures for predictive modeling. Results of your model can always be converted for presentation!

Let's wrap up now on converting the **season** variable into a more interpretable format:

```
# Convert level values for season variable
season_mapping = {0:'spring',
                  1:'summer',
                  2:'fall',
                  3:'winter'}

bikesharing['season'] = bikesharing.season.map(season_mapping)

season_cats = ['spring', 'summer', 'fall', 'winter']
bikesharing['season'] = bikesharing['season'].astype('category')

# Inspect season
print(bikesharing['season'].head())
print(bikesharing['season'].value_counts())

# Visualize distribution of observations across seasons
plt.hist(bikesharing['season'])
plt.title('Bikeshare Distribution Across Seasons')
plt.xlabel('Seasons')
plt.ylabel('Number of Observations')
plt.show()
```

Output of print statements:

```
timestamp
2015-01-04 00:00:00    winter
2015-01-04 01:00:00    winter
2015-01-04 02:00:00    winter
2015-01-04 03:00:00    winter
2015-01-04 04:00:00    winter
Name: season, dtype: category

Categories (4, object): [fall, spring, summer, winter]
spring    4394
summer    4387
winter    4330
fall      4303
Name: season, dtype: int64
```



Well it seems are data is quite balanced across all four seasons! That's a great conclusion as it's one less imbalanced column we would have to handle if we were to perform predictive modeling! Alright let's have one last look at our new data set and move onto some data exploration!

```
# View new data set:
print(bikesharing.head())
print(bikesharing.info())
```

Output of print statements:

```
                     cnt    t1    t2  ...  season  ride_condition  wind_speed_mph
timestamp                             ...
2015-01-04 00:00:00  182  37.4  35.6  ...  winter            fair        3.729024
2015-01-04 01:00:00  138  37.4  36.5  ...  winter            good        3.107520
2015-01-04 02:00:00  134  36.5  36.5  ...  winter            good        0.000000
2015-01-04 03:00:00   72  35.6  35.6  ...  winter            good        0.000000
2015-01-04 04:00:00   47  35.6  32.0  ...  winter            good        4.039776

[5 rows x 11 columns]
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 17414 entries, 2015-01-04 00:00:00 to 2017-01-03 23:00:00
Data columns (total 11 columns):
cnt               17414 non-null int64
t1                17414 non-null float64
t2                17414 non-null float64
hum               17414 non-null float64
wind_speed        17414 non-null float64
weather_code      17414 non-null float64
is_holiday        17414 non-null bool
is_weekend        17414 non-null bool
season            17414 non-null category
ride_condition    17414 non-null category
wind_speed_mph    17414 non-null float64
dtypes: bool(2), category(2), float64(6), int64(1)
memory usage: 1.1 MB
None
```

4 Data Exploration

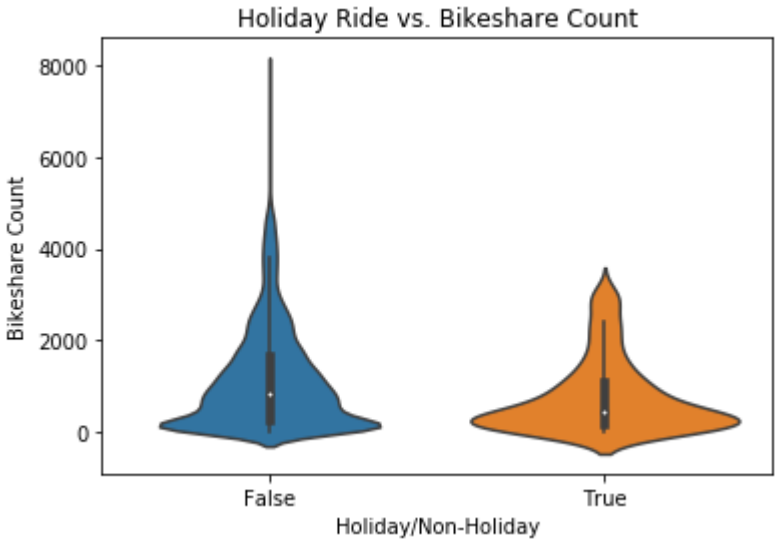
Alright we covered a lot so far! Let's move right into some exploratory data analysis! First let's review some statistical summaries of the data. You may have noticed results were cut off above while using the *.head()* method. I personally like to see a list of summaries for each variable especially since there aren't too many. I'll supply the code below, but due to the amount of output I will not display that here:

```
for var in bikesharing:
    print('Variable:\n ', var, bikesharing[var].describe())
```

4.1 Visualizing Holiday & Weekend Rides

Wow the max value for new bikeshare count is significantly higher than the interquartile range for the column! We're going to learn more about this outlier through the following plots. We've been working a lot with categorical and boolean features, so let's visualize their individual impacts on bike share counts using seaborn. Below we'll produce a few violin plots. Violin plots provide more insight on distribution than your ordinary boxplot:

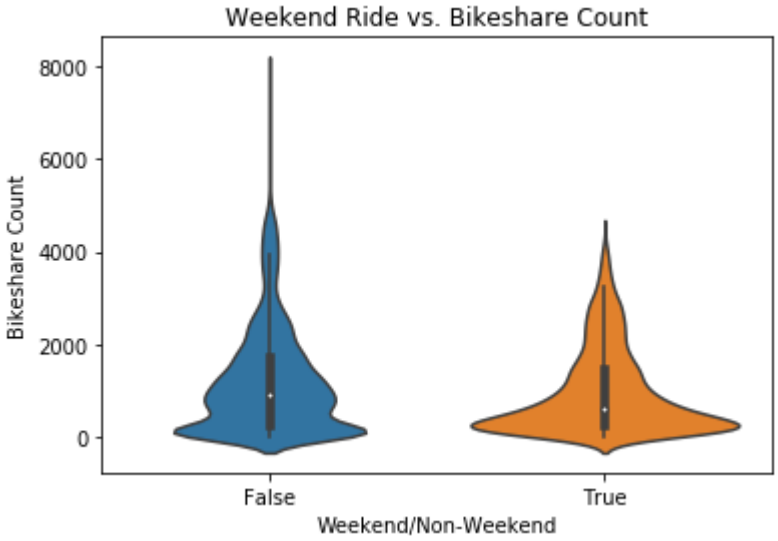
```
# Violin plot for is_holiday
sns.violinplot(x='is_holiday', y='cnt', data=bikesharing)
plt.title('Holiday Ride vs. Bikeshare Count')
plt.xlabel('Holiday/Non-Holiday')
plt.ylabel('Bikeshare Count')

# Mean and median of cnt by is_holiday
# Focus on median due to presence of outliers
bikesharing.groupby('is_holiday')['cnt'].median()
```




```
# Median of cnt vs is_holiday:
is_holiday
False    855.0
True     439.5
Name: cnt, dtype: float64
```

Hmm the bikeshare spike did not seem to occur on a holiday! It seems that the average number of bikeshares is lower for holidays than it is for non-holidays. Let's see if the spike occurred on a weekend day:

```
# Violin plot for is_weekend
sns.violinplot(x='is_weekend', y='cnt', data=bikesharing)
plt.title('Weekend Ride vs. Bikeshare Count')
plt.xlabel('Weekend/Non-Weekend')
plt.ylabel('Bikeshare Count')

# Median of cnt by is_weekend
bikesharing.groupby('is_weekend')['cnt'].median()
```

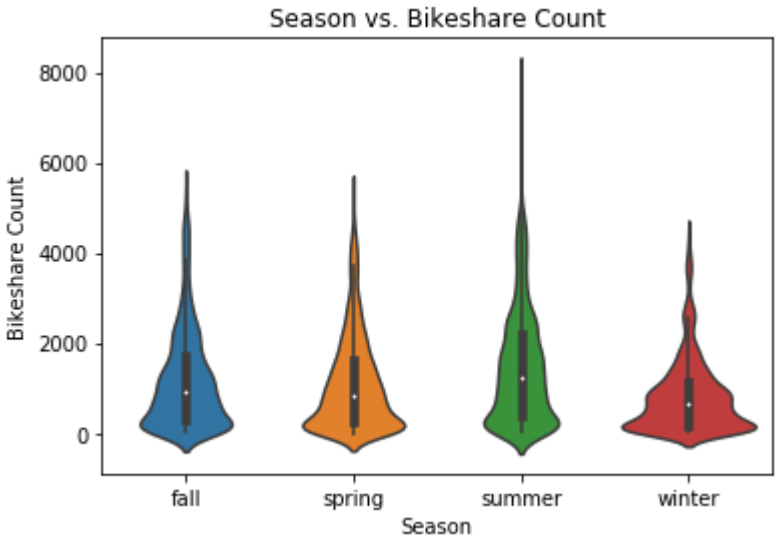



```
# Median of cnt vs is_weekend:
is_weekend
False    927
True     619
Name: cnt, dtype: int64
```

This happened during the week on a day that wasn't a holiday! What was the season of the spike?

4.2 Visualizing Effects of Ride Condition and Season on New Bike Share Counts

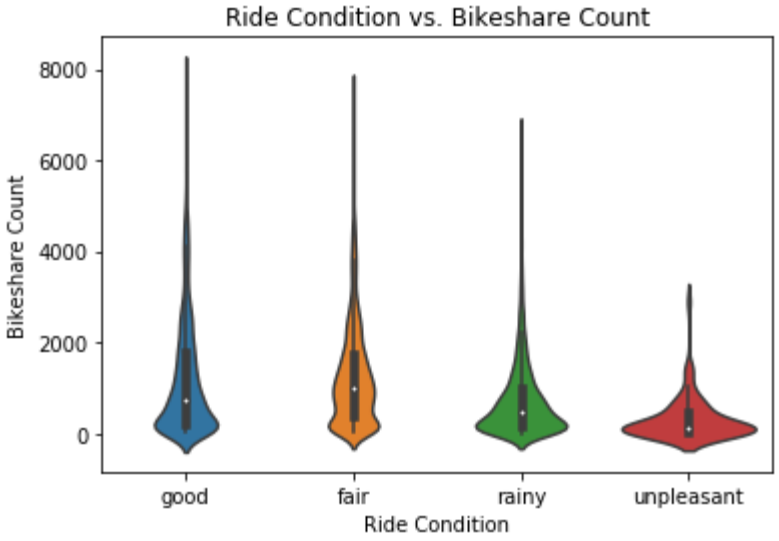
```
# Violin plot for season
sns.violinplot(x='season', y='cnt', data=bikesharing)
plt.title('Season vs. Bikeshare Count')
plt.xlabel('Season')
plt.ylabel('Bikeshare Count')

# Mean and median of cnt by season
bikesharing.groupby('season')['cnt'].median()
```




```
# Median of cnt vs season
season
fall       898
spring     823
summer    1214
winter     632
Name: cnt, dtype: int64
```

Looks like the spike occured during the summer. While we're at it let's investigate how the ride condition was that day:

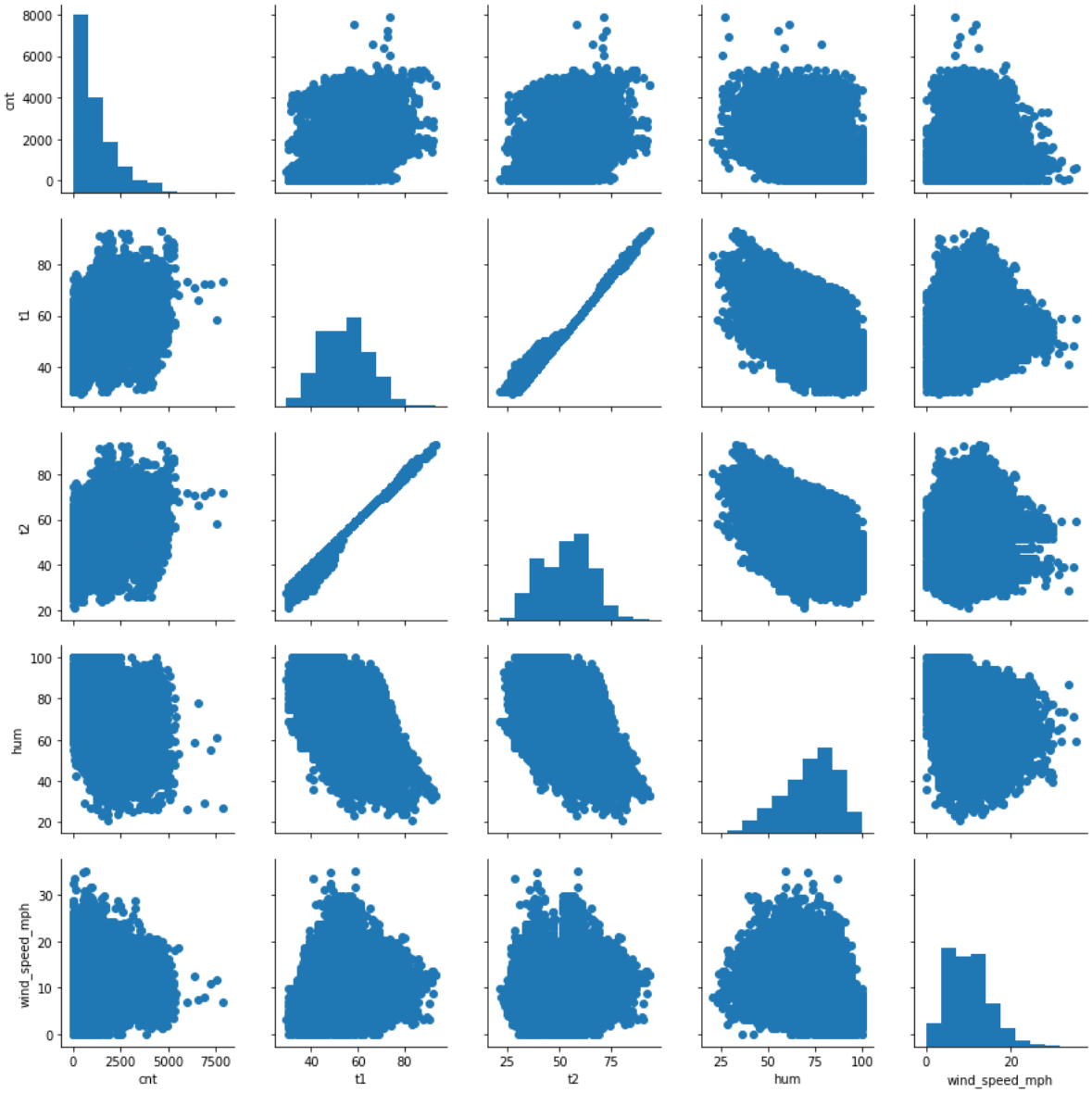
```
# Violin plot for ride_condition
sns.violinplot(x='ride_condition', y='cnt', data=bikesharing)
plt.title('Ride Condition vs. Bikeshare Count')
plt.xlabel('Ride Condition')
plt.ylabel('Bikeshare Count')

# Median of cnt by ride_condition
bikesharing.groupby('ride_condition')['cnt'].median()
```



```
# Median of cnt vs ride_condition:
ride_condition
good           745
fair          1013
rainy          471
unpleasant     122
Name: cnt, dtype: int64
```

Lastly, let's check the relationships between the float64 dtype variables:

```
# Grid plot of numeric variables
p = sns.PairGrid(data=bikesharing, vars=['cnt','t1', 't2', 'hum', 'wind_speed_mph'])
p.map_diag(plt.hist)
p.map_offdiag(plt.scatter)
plt.show()
```



There doesn't seem to be a strong presence of correlation between the features and response variable. There is, however a strong presence of colinearity between **t1** & **t2**. This makes sense based on the provided

variable descriptions. There is also a presence of colinearity between **t1** & **hum**, as well as for **t2** & **hum**. This again makes sense based on the relationship between temperature and humidity.

4.3 Time Series on New Bikeshare Count Spike

It looks like it was a good day for riding as riding conditions were good. Let's look into the date of this spike. With some cross-referential research we can check if there was a major event that occured this day. First we need to slice our dataset for the day this spike occured:

```
# Slicing max new bikeshare count row
bikeshare_spike = bikesharing[bikesharing['cnt'] == max(bikesharing['cnt'])]
print(bikeshare_spike)
```

Output of print statement:

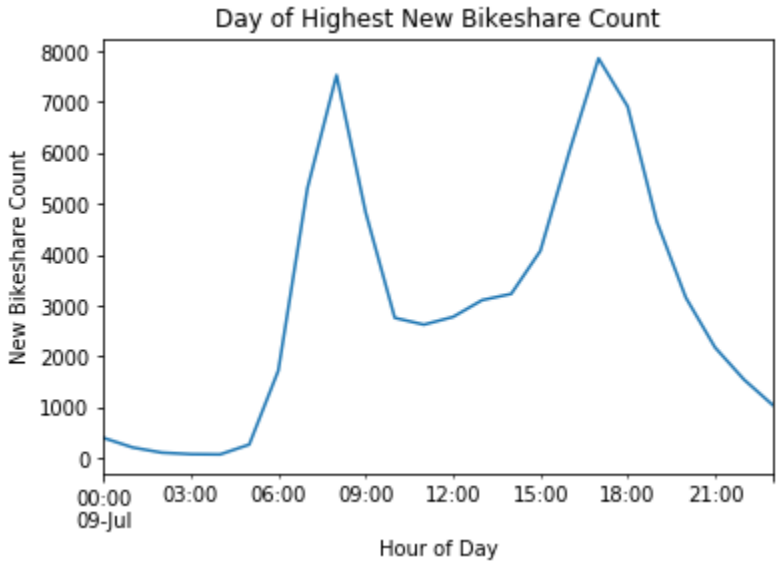
```
                     cnt    t1    t2  ...  season  ride_condition  wind_speed_mph
timestamp                               ...
2015-07-09 17:00:00  7860  73.4  71.6  ...  summer            good        6.836544

[1 rows x 11 columns]
```

Now that we know the date, let's grab all rows associated with this date and plot the rows:

```
# Slicing date of occurence
day_of_spike = bikesharing.loc['2015-07-09']

# Create plot
day_of_spike.cnt.plot()
plt.title('Day of Highest New Bikeshare Count')
plt.xlabel('Hour of Day')
plt.ylabel('New Bikeshare Count')
plt.show()

# Times of spikes
day_of_spike.cnt.sort_values(ascending=False)[0:2]
```

Output of sort method:

```
timestamp
2015-07-09 17:00:00    7860
2015-07-09 08:00:00    7531
Name: cnt, dtype: int64
```

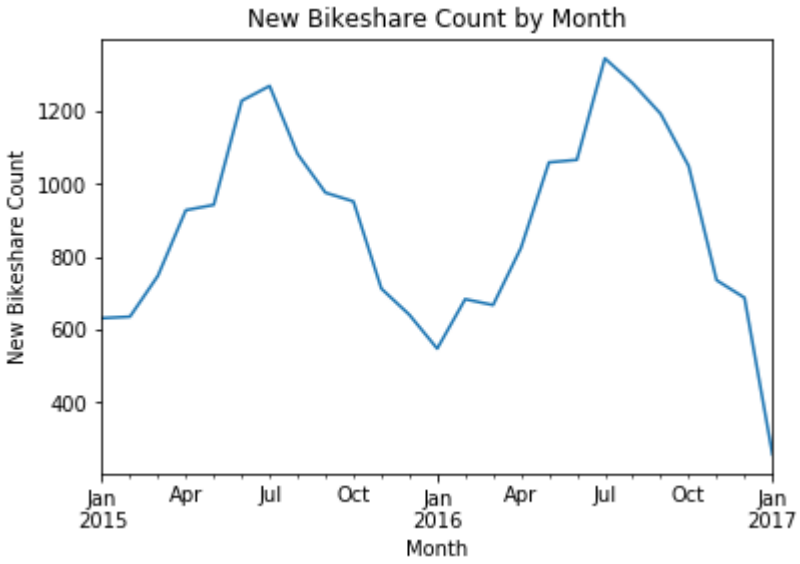


The data reveals that there were actually two spikes that particular day! These occurred at 8am & 5pm. The only major world event I could research was that July 9th, 2015 was the day Chuck Blazer was officially banned from the FIFA organization for corruption. Feel free to do more research of your own to try and reason for this spike!

4.4 Time Series on New Bikeshare Count by Month

Let's resample the data to investigate how new bikeshare counts fluctuate across each month:

```
# Resample data by month
# Note: We use median here because of the presence of outliers
bikeshares_month = bikesharing.resample('M').median()

# Visualize trends in new bikeshare count by month

bikeshares_month.cnt.plot()
plt.title('New Bikeshare Count by Month')
plt.xlabel('Month')
plt.ylabel('New Bikeshare Count')
plt.show()
```



4.5 A Study on Bikesharing using Pivot Tables

Now let's use a pivot table to gain more insights on how various conditions effect counts of new bikeshares. In this section we will establish a pivot table that has the columns **season** and **ride_condition** as rows, **is_holiday** & **is_weekend** as columns, values respectably being **cnt**, and the aggregate function will be 'count' for this table. Here is how it works:

```
# Should remove index for pivot tables
bikeshares_pt = bikesharing.reset_index()

bikeshares_pt.pivot_table(index=['season', 'ride_condition'],
                          columns=['is_holiday', 'is_weekend'],
                          values='cnt',
                          aggfunc='count')
```

Here is the pivot table produced:

```
is_holiday              False         True
is_weekend              False  True   False
season ride_condition
fall   good             930.0  426.0    NaN
       fair            1766.0  651.0    NaN
       rainy            377.0  142.0    NaN
       unpleasant         4.0    7.0    NaN
spring good            1296.0  425.0   31.0
       fair            1300.0  665.0  120.0
       rainy            341.0  163.0   40.0
       unpleasant        10.0    2.0    1.0
summer good            1292.0  523.0    7.0
       fair            1461.0  623.0   26.0
       rainy            335.0  100.0   15.0
       unpleasant         4.0    1.0    NaN
winter good             810.0  305.0  105.0
       fair            1723.0  691.0   23.0
       rainy            385.0  227.0   16.0
       unpleasant        26.0   19.0    NaN
```

We can see that the highest counts of new bikeshares surprisngly occured in the fall and winter during fair riding conditions on non weekend or holiday dates. It appears new bike shares were more likely to occur during unpleasant riding conditions during the spring and winter than in the spring and summer.

5 Statistical Exploratory Data Analysis

In this section we will continue to explore the bikesharing data set. So far most of the data analysis has focused around the categorical variables. Let's further explore the numeric data in the set starting with visualizing each of the variable's empirical cumulative density functions.

5.1 Grabbing Numeric Variables from DataFrame

```
# Grab desired numeric variables for machine learning
desired_numeric_vars = ['cnt', 't1', 't2', 'hum', 'wind_speed']

# Note: Selection could have simply been done by just using the line above, but below shows how
# to grab columns based on datatype
bikeshare_numeric_vars = bikesharing.select_dtypes(include = ['int64', 'float64'])
bikeshare_numeric_vars = bikeshare_numeric_vars[desired_numeric_vars]
```

5.2 Compute ECDF's for Numeric Variables

See appendix for ecdf() function:

```
# ECDF's
bikeshare_ecdf_grab = {}
bikeshare_ecdf_dict = {}

for vars in bikeshare_numeric_vars.columns:
    bikeshare_ecdf_grab['{}_x_set'.format(vars)], bikeshare_ecdf_grab['{}_y_set'.format(vars)] =
  ecdf(bikeshare_numeric_vars[vars])
    bikeshare_ecdf_dict[str(vars)] = bikeshare_ecdf_grab
    bikeshare_ecdf_grab={}
```

5.3 Compute Theoretical Normal PDF for Numeric Variables

In this section we will have to compute the mean and standard deviation of each numeric variable in order to obtain a normal distribution sample with respect to each numeric variable. Here we will be obtaining 10,000 normal samples for each variable

```
# Means
bikeshare_means = {}

for vars in bikeshare_numeric_vars.columns:
    bikeshare_means[str(vars)] = np.mean(bikeshare_numeric_vars[vars])

# Standard Deviations
bikeshare_stds = {}

for vars in bikeshare_numeric_vars.columns:
    bikeshare_stds[str(vars)] = np.std(bikeshare_numeric_vars[vars])

### Samples from normal distribution with respect to given variables' mean and std
bikeshare_numericvars_normalsamp = {}

for vars in bikeshare_numeric_vars.columns:
    bikeshare_numericvars_normalsamp['{}_sample'.format(vars)] = \
np.random.normal(bikeshare_means[str(vars)], bikeshare_stds[str(vars)], 10000)
```

Now we will use the same ecdf() formula to calculate the theoretical normal pdf's for the variables in the same fashion as before.

```
### Theoretical PDF's for variables
bikeshare_theor_grab = {}
bikeshare_theor_dict = {}

for vars in bikeshare_numeric_vars.columns:
    bikeshare_theor_grab['{}_xtheor_set'.format(vars)], bikeshare_theor_grab['{}_ytheor_set'.format(vars
    ecdf(bikeshare_numericvars_normalsamp['{}_sample'.format(vars)])
    bikeshare_theor_dict[str(vars)] = bikeshare_theor_grab
    bikeshare_theor_grab={}
```

5.4 Calculate Percentiles

```
# Percentiles
percentiles = np.array([2.5, 25, 50, 75, 97.5])

bikeshare_percentiles = {}

for vars in bikeshare_numeric_vars.columns:
    bikeshare_percentiles[str(vars)] = np.percentile(bikeshare_numeric_vars[vars], percentiles)

bikeshare_percentiles
```

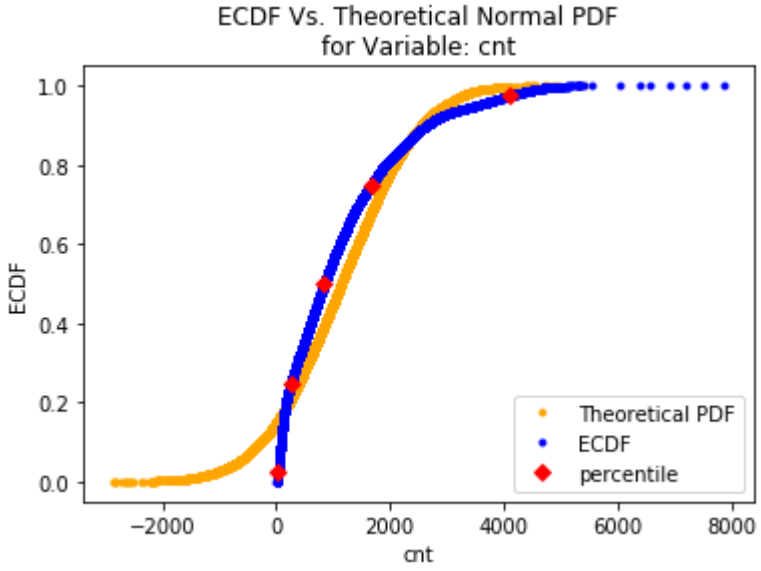
Bikeshare Percentiles:

```
{'cnt': array([  40.  ,  257.  ,  844.  , 1671.75, 4109.  ]),
 't1': array([ 2.5,  8. , 12.5, 16. , 23.5]),
 't2': array([-0.5,  6. , 12.5, 16. , 23.5]),
 'hum': array([41. , 63. , 74.5, 83. , 94. ]),
 'wind_speed': array([ 4. , 10. , 15. , 20.5, 33. ])}
```

5.5 Plot ECDF's Vs Theoretical Normal PDF's with Overlayed Percentiles

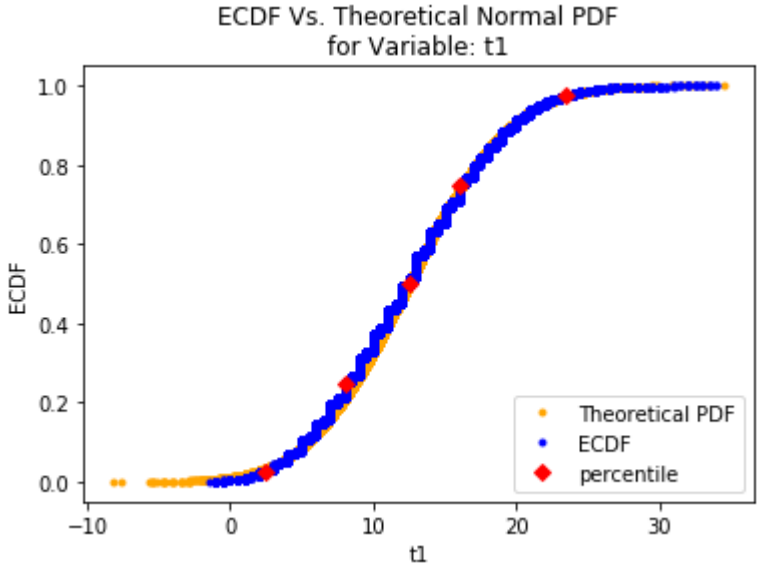
See appendix for ecdf_dict_to_plot():

```
# Plotting bikeshare ECDF's
for vars in bikeshare_ecdf_dict.keys():
    ecdf_dict_to_plot(bikeshare_ecdf_dict,
                      bikeshare_theor_dict,
                      str(vars),
                      bikeshare_percentiles)
    plt.show()
    print('{} percentiles: '.format(vars), bikeshare_percentiles[str(vars)])
```

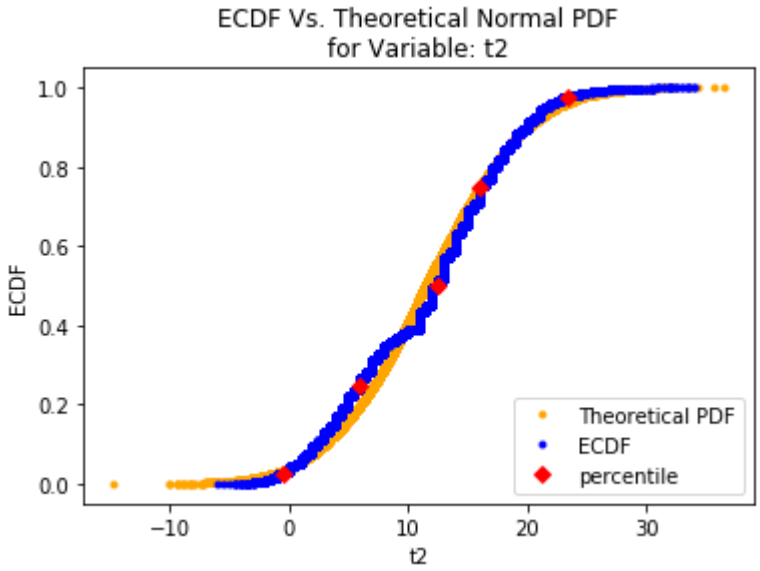



```
cnt percentiles:  [  40.    257.    844.   1671.75 4109.  ]\\
```

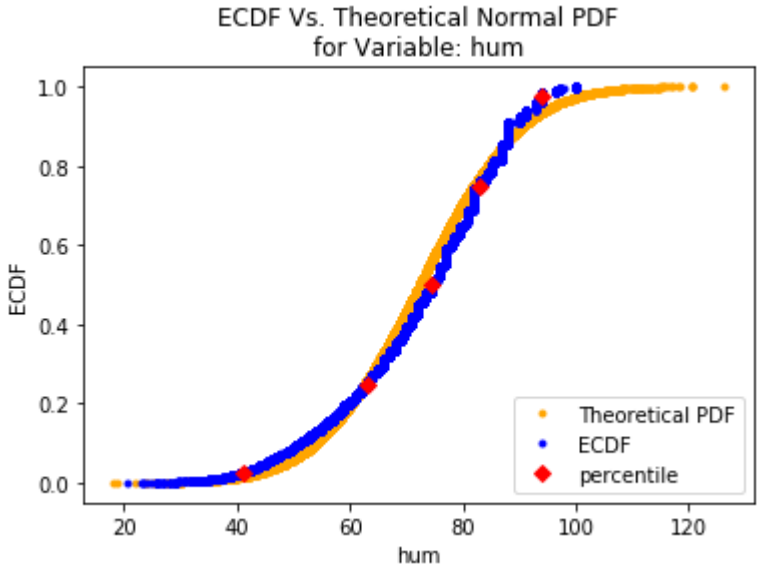
Note: It is clear that **cnt** does not follow a normal distribution. We will want to explore more into what distribution this function follows.



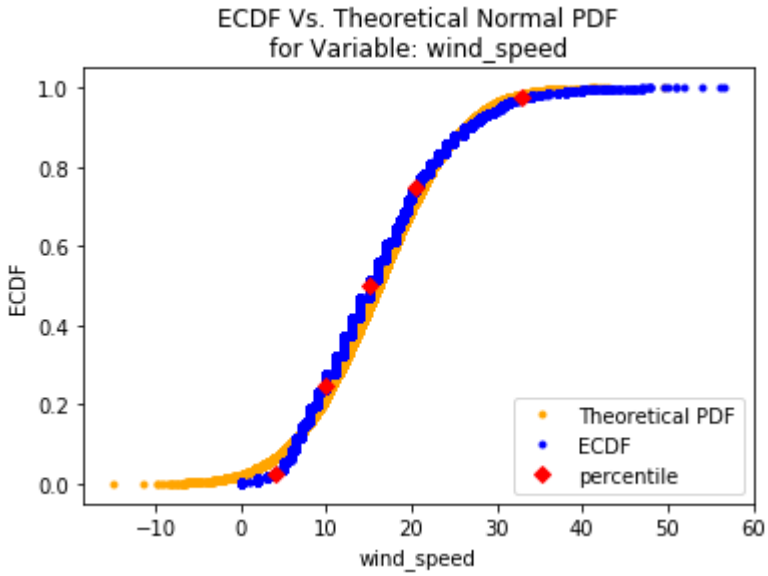

```
t1 percentiles:  [ 2.5  8.  12.5 16.  23.5]
```




```
t2 percentiles:  [-0.5  6.  12.5 16.  23.5]
```




```
hum percentiles:  [41.  63.  74.5 83.  94. ]
```




```
wind_speed percentiles:  [ 4.  10.  15.  20.5 33. ]
```

5.6 Compute Univariate Variances

```
# Campute variances
bikeshare_variance = {}

for vars in bikeshare_numeric_vars.columns:
    bikeshare_variance[str(vars)] = np.var(bikeshare_numeric_vars[vars])

bikeshare_variance
```

Bikeshare variances:

```
{'cnt': 1177391.9034163882,
 't1': 31.043368184804475,
 't2': 43.75762611779025,
 'hum': 204.8555316821771,
 'wind_speed': 62.320661711947885}
```

5.7 Compute Bivariate Covariances

```
bikeshare_covariances = {}
variable_cov_matrix = []

for var1 in bikeshare_numeric_vars.columns:
    for var2 in bikeshare_numeric_vars.columns:
        if str(var1) == str(var2):
            next
        else:
            variable_cov_matrix = np.cov(bikeshare_numeric_vars[var1], bikeshare_numeric_vars[var2])
            bikeshare_covariances['{}_{}_cov'.format(var1, var2)] = variable_cov_matrix[1, 0]

# Filter covariances of features with respect to response variable
bikeshare_cnt_covs = {k:v for (k, v) in bikeshare_covariances.items() if k.startswith('cnt')}
```

Feature covariances with respect to response variable:

```
{'cnt_t1_cov': 2350.684841551497,
 'cnt_t2_cov': 2648.9859465589834,
 'cnt_hum_cov': -7189.478617623968,
 'cnt_wind_speed_cov': 996.2386748248172}
```

5.8 Compute Pearson's Correlation Coefficient

See appendix for pearson_r():

```
bikeshare_corr_coefs = {}

for var1 in bikeshare_numeric_vars.columns:
    for var2 in bikeshare_numeric_vars.columns:
        if str(var1) == str(var2):
            next
        else:
            bikeshare_corr_coefs['{}_{}_cor'.format(var1, var2)] = \
pearson_r(bikeshare_numeric_vars[var1], bikeshare_numeric_vars[var2])

# Filter correlation coefficients of features with respect to response variable
bikeshare_cnt_corrs = {k:v for (k, v) in bikeshare_corr_coefs.items() if k.startswith('cnt')}
```

Feature correlation coefficient's with respect to response variable:

```
{'cnt_t1_cor': 0.38879845125473067,
 'cnt_t2_cor': 0.3690347940673456,
 'cnt_hum_cor': -0.4629009648427378,
 'cnt_wind_speed_cor': 0.11629523130937683}
```

5.9 Closer Examination of Outliers

In this section we will determine the presence of *mild* & *extreme* outliers. See appendix for univariate_fences().

```
# Calculate fences for cnt
cnt_fences = univariate_fences(bikeshare_percentiles['cnt'])

# Grab observations containing mild outliers
cnt_mild_outliers = bikesharing[((bikesharing['cnt'] >= cnt_fences['upper_inner']) & \
                                 (bikesharing['cnt'] < cnt_fences['upper_outer'])) |
                                 ((bikesharing['cnt'] <= cnt_fences['lower_inner']) &
                                  (bikesharing['cnt'] > cnt_fences['lower_outer']))]

# Grab extreme outliers
cnt_extreme_outliers = bikesharing[(bikesharing['cnt'] >= cnt_fences['upper_outer']) |
                                  (bikesharing['cnt'] <= cnt_fences['lower_outer'])]

# Print:
cnt_fences
print(cnt_mild_outliers.shape)
print(cnt_mild_outliers.head())
print(cnt_extreme_outliers.shape)
print(cnt_extreme_outliers.head())
```

Output of print statements (Note: you might notice a new variable called 'rush_hour', more will be covered on this in the next section):

```
# Fences
{'lower_outer': -3987.25,
 'lower_inner': -1865.125,
 'upper_inner': 3793.875,
 'upper_outer': 5916.0}

# Mild Outliers
(668, 14)
                     cnt   t1   t2  ...  t2_F  wind_speed_mph  rush_hour
timestamp                           ...
2015-01-13 08:00:00  3960  9.0  6.5  ...  43.7        9.944065       True
2015-01-14 08:00:00  3904  4.0 -1.0  ...  30.2       16.159105       True
2015-01-22 08:00:00  3828  2.0  1.0  ...  33.8        3.107520       True
2015-01-27 08:00:00  3835  5.5  2.5  ...  36.5        9.322561       True
2015-02-24 08:00:00  3841  6.0  2.0  ...  35.6       12.430081       True

[5 rows x 14 columns]
# Extreme Outliers
(7, 14)
                     cnt    t1    t2  ...  t2_F  wind_speed_mph  rush_hour
timestamp                             ...
2015-07-09 08:00:00  7531  14.5  14.5  ...  58.1       11.808577       True
2015-07-09 16:00:00  6033  23.0  22.0  ...  71.6        6.836544      False
2015-07-09 17:00:00  7860  23.0  22.0  ...  71.6        6.836544       True
2015-07-09 18:00:00  6913  22.5  21.5  ...  70.7        8.079553       True
2015-08-06 08:00:00  6585  19.0  19.0  ...  66.2        7.458048       True

[5 rows x 14 columns]
```

Interestingly enough, the 49.7% of the mild outliers occurred at 8am, and 24.9% occurred at 5pm. This is illustrated below:

```
# 8am occurrences:
cnt_mild_outliers_8am = cnt_mild_outliers[cnt_mild_outliers.index.hour == 8]
len(cnt_mild_outliers_8am)/len(cnt_mild_outliers)

# 5pm occurrences:
cnt_mild_outliers_5pm = cnt_mild_outliers[cnt_mild_outliers.index.hour == 17]
len(cnt_mild_outliers_5pm)/len(cnt_mild_outliers)
```

Output:

```
# 8am occurrences:
0.49700598802395207

# 5pm occurrences:
0.24850299401197604
```

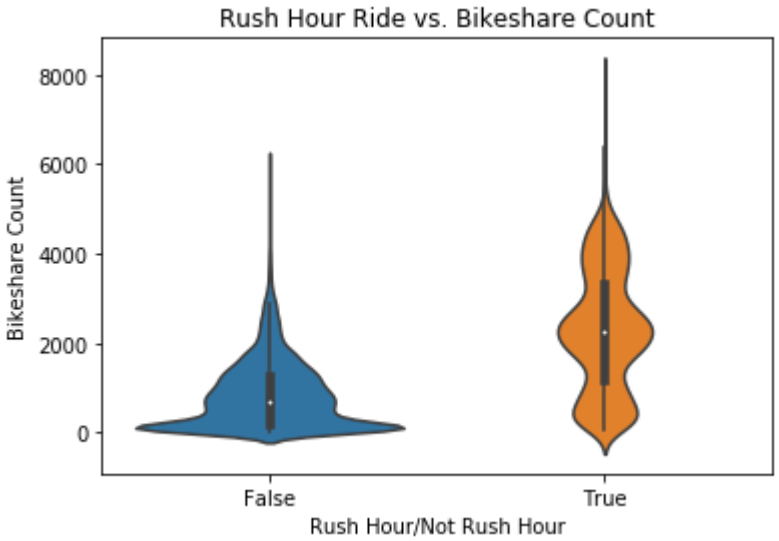
There's no denying that rush hour has a significant impact on new bikeshare count. We must further consider this impact as the focus of the project is to build the best prediction model for new bikeshare count. Let's create a new variable called **rush_hour**.

```
# Create new variable: rush_hour
bikesharing['rush_hour'] = False

for i in range(len(bikesharing.index.hour)):
```

```
        if ((bikesharing.index.hour[i] >= 7) & \
  (bikesharing.index.hour[i] < 10)) == True:
            bikesharing['rush_hour'][i] = True
        elif ((bikesharing.index.hour[i] >= 17) & \
      (bikesharing.index.hour[i] < 19)) == True:
            bikesharing['rush_hour'][i] = True
        else:
            bikesharing['rush_hour'][i] = False
```

Let's explore the characteristics of this new variable:

```
# Explore characteristics of new variable
sns.violinplot(x='rush_hour', y='cnt', data=bikesharing)
plt.title('Rush Hour Ride vs. Bikeshare Count')
plt.xlabel('Rush Hour/Not Rush Hour')
plt.ylabel('Bikeshare Count')
plt.show()

print(bikesharing.rush_hour.value_counts())

# Median & mean of cnt by is_weekend
bikesharing.groupby('rush_hour')['cnt'].median()
bikesharing.groupby('rush_hour')['cnt'].mean()
```



Output of print & groupby statements:

```
# Value counts:
False    13781
True      3633

# cnt median vs rush_hour
rush_hour
False     679
True     2265
Name: cnt, dtype: int64

# cnt mean vs rush_hour
rush_hour
```

```
False     840.060373
True     2292.623176
Name: cnt, dtype: float64
```

There is a significant difference between means & medians of **cnt** with respect to **rush_hour**. This shows that **rush_hour** will be a significant contributor to our machine learning model.

How many of both the *mild* & *extreme* outliers can be explained by the variable **rush_hour**?

```
# Compute number & proportion of mild & extreme outliers occurring during rush hour
# Update cnt_mild_outliers & cnt_extreme_outliers
cnt_mild_outliers['rush_hour'] = bikesharing['rush_hour']
cnt_extreme_outliers['rush_hour'] = bikesharing['rush_hour']

rush_hour_ride = cnt_mild_outliers.rush_hour.sum()
extreme_rush_hour_ride = cnt_extreme_outliers.rush_hour.sum()

print('Number of Mild Outliers During Rush Hour: {}'.format(rush_hour_ride))
print('Proportion of Mild Outliers During Rush Hour: {}'. \
format(rush_hour_ride/cnt_mild_outliers.shape[0]))
print('Number of Extreme Outliers During Rush Hour: {}'. \
format(extreme_rush_hour_ride))
print('Proportion of Extreme Outliers During Rush Hour: {}'. \
format(extreme_rush_hour_ride/cnt_extreme_outliers.shape[0]))
```

Output of print statements:

```
Number of Mild Outliers During Rush Hour: 632
Proportion of Mild Outliers During Rush Hour: 0.9461077844311377
Number of Extreme Outliers During Rush Hour: 6
Proportion of Extreme Outliers During Rush Hour: 0.8571428571428571
```

By now you should be able to see that if we simply deleted all of the outliers based on the upper inner fence we would have lost significant amounts of information about our data set. Even the majority of the extreme outliers can be explained by **rush_hour**. Lastly, let's take a glance at the one *extreme* outlier not explained by **rush_hour**.

```
# Explore values of observation containing extreme & mild outlier not occuring during rush hour
odd_extreme_outlier = cnt_extreme_outliers[cnt_extreme_outliers['rush_hour'] == False]

print(odd_extreme_outlier)
```

Output of print statement:

```
                     cnt    t1    t2  ...  t2_F  wind_speed_mph  rush_hour
timestamp                             ...
2015-07-09 16:00:00  6033  23.0  22.0  ...  71.6        6.836544      False
```

This outlier didn't occur during a holiday or a weekend day. The spike seemed to occur during then summer. With further statistical inference we may be able to conclude the omition of this outlier.

6 Probabilistic Reasoning on 'cnt'

6.1 Likelihood of Reoccurence of Outlier values

In this section we will explore the likelihood of observing the smallest outlier value for cnt or greater again. We will use the smallest outlier value as it reasonably has the highest likelihood of occuring again. We can assume that probability reduces as outlier values increase. Here the value is **3794**.

Note: For this portion we will assume that there is a statistically significant difference in population means for **rush_hour** being *True* or *False*. For now we will also assume cnt does follow a normal distribution.

```
# Extracting smallest outlier value
 cnt_smallest_mild_outlier = min(cnt_mild_outliers.cnt)

# Sample mean and standard deviation to draw theoretical samples from
mu = bikesharing.cnt.mean()
sigma = bikesharing.cnt.std()

# Draw 1M normally distributed samples
samples = np.random.normal(mu, sigma, 1000000)

# Compute fraction of new bikeshare cnt's >= 3794
prob = np.sum(samples >= 3794)/len(samples)

print('Likelihood of New Bikeshare count being >= to 3794: {}'.format(prob))
```

Output of print statement:

```
Likelihood of New Bikeshare count being >= to 3794: 0.007431
```

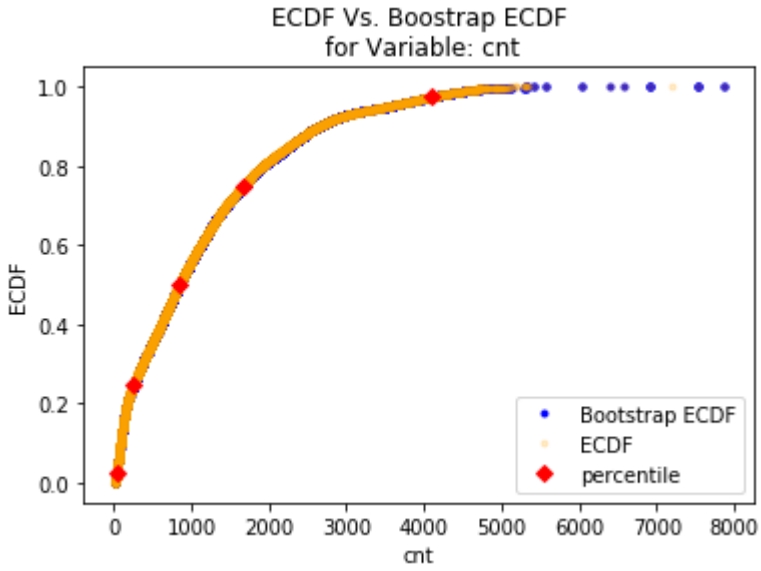
It looks like there is a 0.74% chance of witnessing an observation for **cnt** of 3794 or greater. This appears to be a rare event.

6.2 Comparing Bootstrap ECDF to Sample's ECDF

In this section we will generate bootstrap ECDF samples and plot these against the observed ECDF. This will illustrate the likelihood of witnessing the observed distribution of **cnt** in a generalized setting.

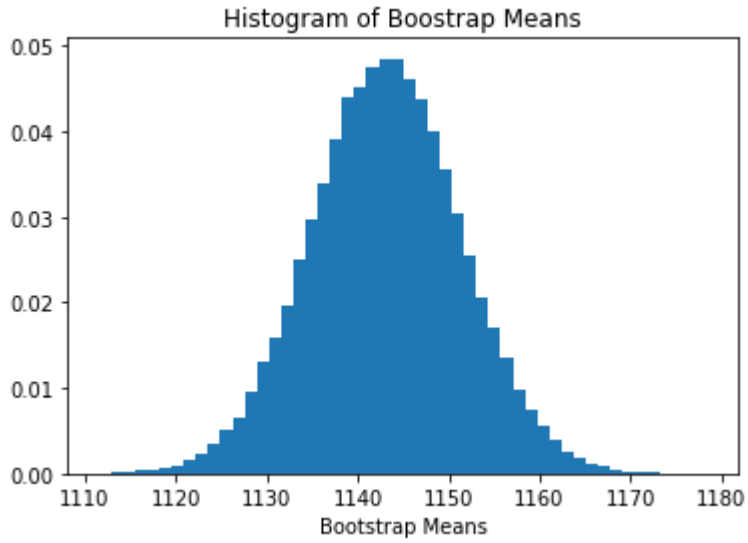
```
# Compare bootstrap ecdf to original ecdf
for b in range(10000):
    bs_sample = np.random.choice(bikesharing.cnt, size=len(bikesharing.cnt))

bs_x, bs_y = ecdf(bs_sample)

# Compute ecdf of original set again
x, y = ecdf(bikesharing.cnt)

# Note: Bootstrap ecdf is almost perfectly in line with original, including outliers
plot = plt.plot(bs_x, bs_y, marker='.', linestyle='none', color='blue')
plot = plt.plot(x, y, marker='.', linestyle='none', color='orange', alpha=0.2)
plot = plt.plot(bikeshare_percentiles['cnt'],
                percentiles/100,
                marker='D',
                color='red',
                linestyle='none')
plot = plt.legend(['Bootstrap ECDF', 'ECDF', 'percentile'], loc='lower right')
```

```
plot = plt.title('ECDF Vs. Boostrap ECDF\n for Variable: cnt')
plot = plt.xlabel('cnt')
plot = plt.ylabel('ECDF')
plt.show()
```



We can see in this plot that the two ECDF's do follow very closely to one another. It is likely that we will witness **cnt** follow this form of distribution in other instances. Next we will compute a 95% confidence interval for **cnt**'s population mean to compare to the sample's mean, compute the standard error of the mean, and plot a histogram of all the boostrap means. See Appendix for draw_bs_reps().

```
# Bootstrap means for cnt
bs_replicates_mean = draw_bs_reps(bikesharing.cnt, np.mean, size=100000)

# Standard Error of Mean
bs_replicates_sem = np.std(bikesharing.cnt)/np.sqrt(len(bikesharing.cnt))

print('Standard Error of Bootstrap Means: {}'.format(bs_replicates_sem))
print('New Bikeshare count sample mean: {}'.format(bikesharing.cnt.mean()))

plot = plt.hist(bs_replicates_mean, bins = 50, normed = True)
plot = plt.title('Histogram of Boostrap Means')
plot = plt.xlabel('Bootstrap Means')
plt.show()
```



Output of print statement:

```
Standard Error of Bootstrap Means: 8.222639542511711
New Bikeshare count sample mean: 1143.1016423567244
```

Standard error is fairly low for the bootstrap mean aggregates of **cnt**. The observed sample mean also seems to follow closely to the center of the boostrap mean distribution. Next we will compute a confidence interval of the boostrap means to confirm this observation:

```
conf_int_mean = np.percentile(bs_replicates_mean, [2.5, 97.5])

print('95% Confidence Interval of Bootstrap Means: {}'.format(conf_int_mean))
```

Output of print statement:

```
95% Confidence Interval of Bootstrap Means: [1127.07164781 1159.29615539]
```

6.3 Hypothesis Testing on Difference of Means for Rush Hour Variable with respect to New Bikeshare Count

In this section we will ask the question: Is there is statistically significant difference in rush hour/ non-rush hour populations? This will be a two-tailed hypothesis test. Our hypotheses will be laid out as the following:

H_0: There is no significant difference in population means for **rush_hour** variable
H_1: There is a significant difference in population means for **rush_hour** variable
Given:

- Mean for Rush Hour = True: 2185.722018
- Standard Deviation for Rush Hour = True: 1286.067033
- Mean for Rush Hour = False: 794.869312
- Standard Deviation for Rush Hour = False: 730.783413

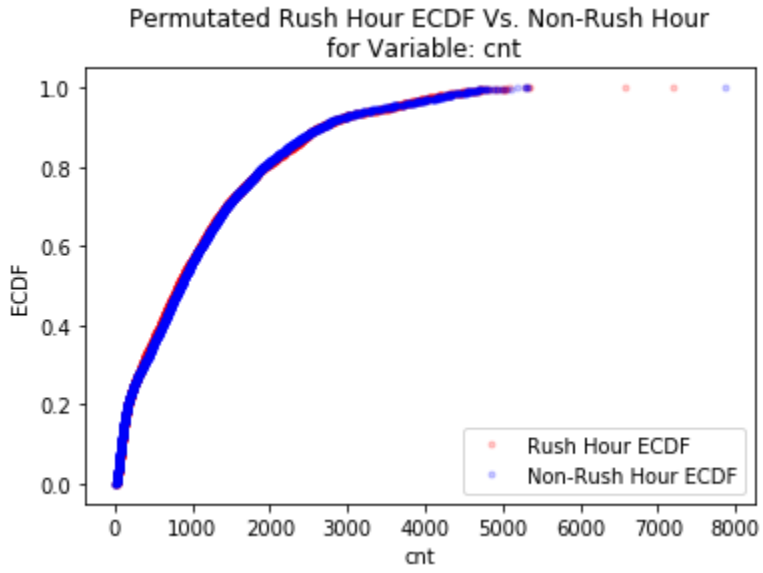
Procedure: Permutation

- cnt column will be permutated
- 4360 observations will be assigned to **rush_hour** = *True* and 13054 observations will be assigned to **rush_hour** = *False*

- Note: Assignment reflects the number of observations containing such values in original data set
- Caution: You may notice these samples are imbalanced; this imbalance will be addressed using another procedure later on

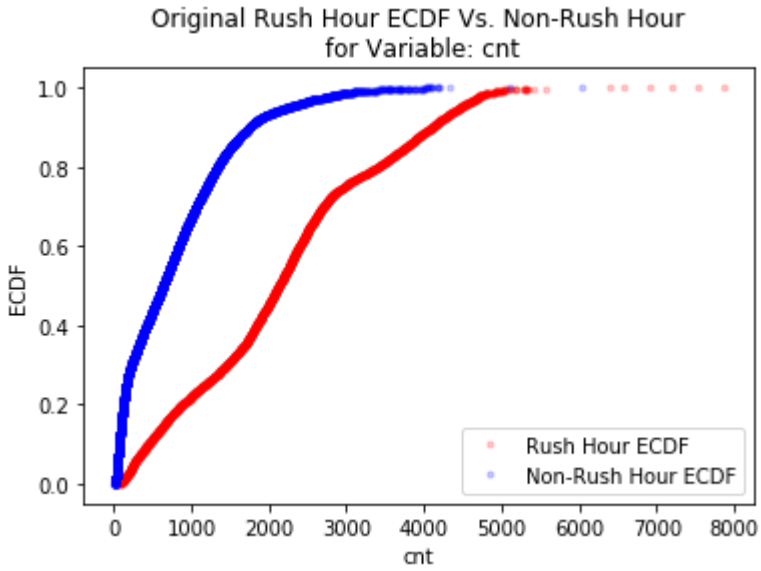
Let's begin by performing the permutation, then comparing ECDF's of the permutated samples to the ECDF's of the original samples for the two populations.

```
# Permutation procedure
cnt_column = bikesharing['cnt']
cnt_column = cnt_column.reset_index(drop=True)

cnt_column_perm = np.random.permutation(cnt_column)

# Define rush hour, non-rush hour samples
rush_hour_perm = cnt_column_perm[:4360]
nonrush_hour_perm = cnt_column_perm[13054:]

# Compute ECDF's
x_1, y_1 = ecdf(rush_hour_perm)
x_2, y_2 = ecdf(nonrush_hour_perm)

# Plot Permutated ECDF's
plot = plt.plot(x_1,
                y_1,
                marker='.',
                linestyle='none',
                color='red',
                alpha=0.2)
plot = plt.plot(x_2,
                y_2,
                marker='.',
                linestyle='none',
                color='blue',
                alpha=0.2)
plot = plt.legend(['Rush Hour ECDF', 'Non-Rush Hour ECDF'], loc='lower right')
plot = plt.title('Permutated Rush Hour ECDF Vs. Non-Rush Hour\n for Variable: cnt')
plot = plt.xlabel('cnt')
plot = plt.ylabel('ECDF')
plt.show()
```



The permuted samples of the two populations suggest that there is no difference in distributions of the two populations. Let's compare this to the instance we witnessed in the original sample:

```
# Compute ECDF's from original data
rush_hour = bikesharing[bikesharing['rush_hour'] == True]
rush_hour_cnt = rush_hour.cnt.reset_index(drop=True)

nonrush_hour = bikesharing[bikesharing['rush_hour'] == False]
nonrush_hour_cnt = nonrush_hour.cnt.reset_index(drop=True)

xo_1, yo_1 = ecdf(rush_hour_cnt)
xo_2, yo_2 = ecdf(nonrush_hour_cnt)

plot = plt.plot(xo_1,
                yo_1,
                marker='.',
                linestyle='none',
                color='red',
                alpha=0.2)
plot = plt.plot(xo_2,
                yo_2,
                marker='.',
                linestyle='none',
                color='blue',
                alpha=0.2)
plot = plt.legend(['Rush Hour ECDF', 'Non-Rush Hour ECDF'], loc='lower right')
plot = plt.title('Original Rush Hour ECDF Vs. Non-Rush Hour\n for Variable: cnt')
plot = plt.xlabel('cnt')
plot = plt.ylabel('ECDF')
plt.show()
```



Our original sample distributions of the two populations tell a different story. Make note here though we may not be witnessing the difference in ECDF's in the permutated illustration as again the number of available observations for both populations are imbalanced. Let's continue onto the hypothesis test. For this test we will be using population mean as our test statistic and p-value. See Appendix for draw_perm_reps():

```
# Difference of Permutated Means
perm_diff = np.mean(rush_hour_perm) - np.mean(nonrush_hour_perm)

# Difference of Original Means
original_diff = np.mean(rush_hour_cnt) - np.mean(nonrush_hour_cnt)

# P-value for probability of permutated means being larger than original
cnt_perm_replicates = draw_perm_reps(rush_hour_cnt, nonrush_hour_cnt, diff_of_means, size=10000)

p = np.sum(cnt_perm_replicates >= original_diff)/len(cnt_perm_replicates)

print('P-value for Difference of Means: {}'.format(p))
```

Output of print statement:

```
P-value for Difference of Means: 0.0
```

Here it seems there is a 0.0% chance of getting a difference of means that is greater than or equal to the observed difference in data set if rush hour and non-rush hour distributions of cnt were the same. There seems to be a significant difference between populations, but this should be taken cautiously due to the present imbalance. Let's use a one-tailed procedure to cross-validate the findings of the hypothesis test.

6.4 Boostrap Approach to Cross-Validate Findings

Here we will use only the mean of the rush hour samples and treat such as the true mean to compare against the non-rush hour samples. This will be a one sample hypothesis test. The hypotheses follow the same as above (we are simply validating our previous findings with a procedure that accounts for the imbalance in present observations for the two populations). More specifically, we are testing for the likelihood of observing a difference of mean less than or equal to what was observed in the sample between the two populations. As part of this procedure we will shift the non-rush hour samples using the rush hour mean:

```
rush_hour_mean = np.mean(rush_hour_cnt)
```

```
# Note: Shift nonrush hour values
nonrush_hour_cnt_shifted = rush_hour_cnt - np.mean(nonrush_hour_cnt) + rush_hour_mean

# Difference from rush hour mean
def diff_from_rush_hour(data, rush_hour_value = 2185.722018348624):
    return np.mean(data) - rush_hour_value

# Observed difference:
obs_diff = diff_from_rush_hour(nonrush_hour_cnt)

# Draw bootstrap replicates
nonrush_bs_replicates = draw_bs_reps(nonrush_hour_cnt_shifted,
                                     diff_from_rush_hour,
                                     10000)
# p-value:
p_value = np.sum(nonrush_bs_replicates <= obs_diff)/10000
print('P-value for Difference of Means: {}'.format(p_value))
```

Output of print statement:

```
P-value for Difference of Means: 0.0
```

Our results hold consistency for both tests. We have shown that there is indeed a statistically significant difference in populations of **cnt** counts that occured during rush hour and non-rush hour times. We can conclude that this variable is indeed a good feature representation of the New Bikeshare count variable.

7 Summary

I hope you enjoyed what has been covered here! Thanks to Python's powerful tools available it's easy to clean data in a swift and organized manner. There is so much more we could explore with data analysis. For instance we could investigate our float64 variables' empirical cumulative distribution function against the samples theoretical cumululative distribution function to see if the variables follow an expected distribution. We could also use tools such as permutation and boostrapping along with hypothesis testing to come up with probabilitistics reasoning, which could help us improve our understanding of the data set before moving into predictive modeling. Thanks for reading!

8 Appendix

Below are the functions created during this project.

8.1 Empirical Cumulative Density Function (ecdf)

```
# Source: DataCamp: Statistical Thinking (Part 1)

def ecdf(data):
    # Number of data points
    n = len(data)

    # x-data for ECDF
    x = np.sort(data)

    # y-data for ECDF
    y = np.arange(1, n+1)/n
```

```
    return x, y
```

8.2 Pearson's Correlation Coefficient

```
# Source: DataCamp: Statistical Thinking (Part 1)

def pearson_r(x, y):
    corr_mat = np.corrcoef(x, y)

    return corr_mat[0, 1]
```

8.3 Determining Mild & Extreme Outliers

```
def univariate_fences(percentiles):
    fences = {}
    IQR = percentiles[3] - percentiles[1]

    fences['lower_outer'] = percentiles[1] - (3 * IQR)
    fences['lower_inner'] = percentiles[1] - (1.5 * IQR)
    fences['upper_inner'] = percentiles[3] + (1.5 * IQR)
    fences['upper_outer'] = percentiles[3] + (3 * IQR)

    return fences
```

8.4 Plotting ECDF

```
def ecdf_dict_to_plot(bikeshare_ecdf_dict,
                      bikeshare_theor_dict,
                      var_name,
                      bikeshare_percentiles):
    """
    The purpose of this function is to plot ECDF's of a DataFrame's numeric
    variables (with overlayed variable percentiles) and compare plots to
    each of the variable's theoretical PDF.
    """

    plot = plt.plot(bikeshare_theor_dict[var_name]['{}_xtheor_set'.format(var_name)],
                                    bikeshare_theor_dict[var_name]['{}_ytheor_set'.format(var_name)],
                                    marker='.',
                                    linestyle='none',
                                    color='orange')

    plot = plt.plot(bikeshare_ecdf_dict[var_name]['{}_x_set'.format(var_name)],
                                    bikeshare_ecdf_dict[var_name]['{}_y_set'.format(var_name)],
                                    marker='.',
                                    linestyle='none',
                                    color='blue')
    # Overlay percentiles
    plot = plt.plot(bikeshare_percentiles[var_name],
                    percentiles/100,
                    marker='D',
                    color='red',
                    linestyle='none')
```

```
    plot = plt.legend(['Theoretical PDF', 'ECDF', 'percentile'], loc='lower right')
    plot = plt.title('ECDF Vs. Theoretical Normal PDF\nfor Variable: {}'.format(var_name))
    plot = plt.xlabel('{}'.format(var_name))
    plot = plt.ylabel('ECDF')


    return plot
```

8.5 Aggregate Bootstrap Replicates in a 1-Dimensional Setting

```
# Source: DataCamp: Statistical Thinking (Part 2)

def bootstrap_replicate_1d(data, func):
    """
    Generate boostrap replicate of 1D data.
    """
    return func(np.random.choice(data, size=len(data)))
```

8.6 Generating Many Bootstrap Aggregates

```
# Source: DataCamp: Statistical Thinking (Part 2)

def draw_bs_reps(data, func, size=1):
    return np.array([bootstrap_replicate_1d(data, func) for _ in range(size)])
```

8.7 Generate Permutation Sample from Two Data Sets

```
# Source: DataCamp: Statistical Thinking (Part 2)

def permutation_sample(data_1, data_2):
    permuted_data = np.random.permutation(np.concatenate((data_1, data_2)))
    return permuted_data[:len(data_1)], permuted_data[len(data_1):]
```

8.8 Generating Many Permutation Replicates

```
# Source: DataCamp: Statistical Thinking (Part 2)

def draw_perm_reps(d1, d2, func, size=1):
    return np.array([func(*permutation_sample(d1, d2)) for i in range(size)])
```

8.9 Difference of Means

```
# Source: DataCamp: Statistical Thinking (Part 2)

def diff_of_means(data_1, data_2):
    return np.mean(data_1) - np.mean(data_2)
```

9 Resources

Source: https://www.visitlondon.com/traveller-information/getting-around-london/london-cycle-hire-scheme
Source: https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset

Source: https://www.cnn.com/2015/07/09/football/football-fifa-blazer-ban/index.html