TORONTO BIKESHARE

Presented to you by The Dilligent Debuggers
December 05, 2022

WHO ARE THE

Diligent Debuggers ?!

Bruce (Haoyu) Li

Fiyad Hussein

Mennatallah Alnahas

Zhijie Nie

TABLE OF CONTENTS

OUR GOALS

Data Exploration

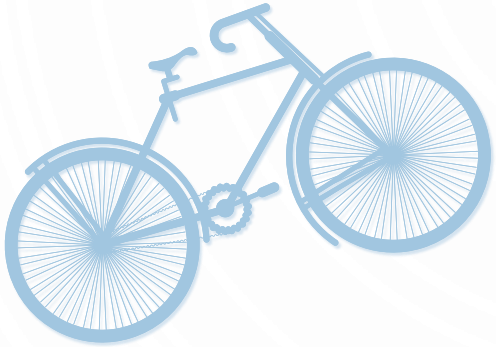
Recognize the temporal and geospatial patterns of bike rides

Ad Campaign

Answer the questions of When and Where?

New App Feature

Propose an intermediate docking station if ride duration > 45 mins



DATASET CLEANING

Data Cleaning-Bikeshare Stations Data

Get
- Retrieve the up-to-date bikeshare station information from `https://tor.publicbikesystem.net`.

Rename
- Rename the columns

Clean
- Check for missing values and duplicates

Output
- Write bikeshare station dataframe to a csv file

```
1  bikeshare_stations.info()
```
[3] ✓ 0.1s Python

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 655 entries, 0 to 654
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Station Id    655 non-null    int32
 1   Station Name  655 non-null    object
 2   lat           655 non-null    float64
 3   lon           655 non-null    float64
 4   capacity      655 non-null    int64
dtypes: float64(2), int32(1), int64(1), object(1)
memory usage: 23.2+ KB
```

Clean station data: “bikeshare_stations”

```
1  bikeshare_stations.nunique()
```
[4] ✓ 0.1s Python

```
Station Id      655
Station Name    655
lat             653
lon             654
capacity         38
dtype: int64
```

Data Cleaning-Weather Data

Load

- Load Weather Data
- Rename the columns

Pre-processing

- Drop unwanted columns
- Check on duplicates
- Datetime Conversion

Clean

- Check for missing values
- Check for outliers

```
1  weather_clean = weather.drop(['Year', 'Month', 'Day', 'Time', 'Station Name', 'Climate ID', 'Longitude (x)', 'Latitude (y)', 'Stn Press (kPa)'], axis=1)
```
✓ 0.9s

```
1  weather_clean[weather_clean.duplicated(subset=['Date/Time'])]
```
✓ 0.1s Python

Date/Time	Temp (°C)	Dew Point Temp (°C)	Rel Hum (%)	Wind Dir (10s deg)	Wind Spd (km/h)	Visibility (km)	Hmdx	Wind Chill	Weather	Precip. Amount (mm)

```
1  # Check missing values for each year
2  display(weather_clean.resample('Y').agg(lambda row: row.isnull().sum()))
```
✓ 0.3s Python

Date/Time	Temp (°C)	Dew Point Temp (°C)	Rel Hum (%)	Wind Dir (10s deg)	Wind Spd (km/h)	Visibility (km)	Hmdx	Wind Chill	Weather	Precip. Amount (mm)
2017-12-31 00:00:00-05:00	11	11	10	530	9	11	7599	7251	7390	8760
2018-12-31 00:00:00-05:00	4	4	2	504	2	4	6979	7084	7336	8760
2019-12-31 00:00:00-05:00	95	145	145	595	23	24	7608	6936	7213	8760
2020-12-31 00:00:00-05:00	13	12	7	426	14	16	7211	7599	7494	8040
2021-12-31 00:00:00-05:00	26	52	47	476	21	16	7307	7383	7624	14
2022-12-31 00:00:00-05:00	234	420	419	292	6	15	4818	4541	5157	4

Data Cleaning-Weather Data

Load

- Load Weather Data
- Rename the columns

Pre-processing

- Drop unwanted columns
- Check on duplicates
- Datetime Conversion

Clean

- **Check for missing values**
- Check for outliers

```
1  weather_clean['Weather'].unique()
```

✓ 0.9s

```
array([nan, 'Fog', 'Rain,Fog', 'Rain', 'Snow', 'Moderate Rain',
       'Moderate Rain,Fog', 'Haze', 'Rain,Snow', 'Freezing Rain,Fog',
       'Snow,Blowing Snow', 'Heavy Snow', 'Moderate Snow',
       'Haze,Blowing Snow', 'Heavy Rain,Fog', 'Freezing Rain,Snow',
       'Thunderstorms,Rain,Fog', 'Freezing Rain', 'Thunderstorms,Rain',
       'Thunderstorms,Moderate Rain,Fog', 'Thunderstorms,Moderate Rain',
       'Thunderstorms', 'Thunderstorms,Heavy Rain,Fog',
       'Thunderstorms,Heavy Rain', 'Thunderstorms,Fog'], dtype=object)
```

```
1  # Modifying missing values in 'Weather' column
2  weather_clean['Weather'].fillna('Clear', inplace = True)
3  weather_clean.head()
```

✓ 0.1s

Linear Interpolation for filling NaN of other numerical

Data Cleaning-Weather Data

Load

- Load Weather Data
- Rename the columns

Pre-processing

- Drop unwanted columns
- Check on duplicates
- Datetime Conversion

Clean

- **Check for missing values**
- Check for outliers

```
1  # Check if there are still missing values
2  weather_clean.isnull().sum()
```

✓ 0.1s

```
Temp (°C)                0
Dew Point Temp (°C)      0
Rel Hum (%)              0
Wind Dir (10s deg)       0
Wind Spd (km/h)          0
Visibility (km)          0
Hmdx                 41522
Wind Chill           40794
Weather                  0
Precip. Amount (mm)      0
dtype: int64
```

Clean weather data: "weather_clean"

Data Cleaning-Weather Data

Load

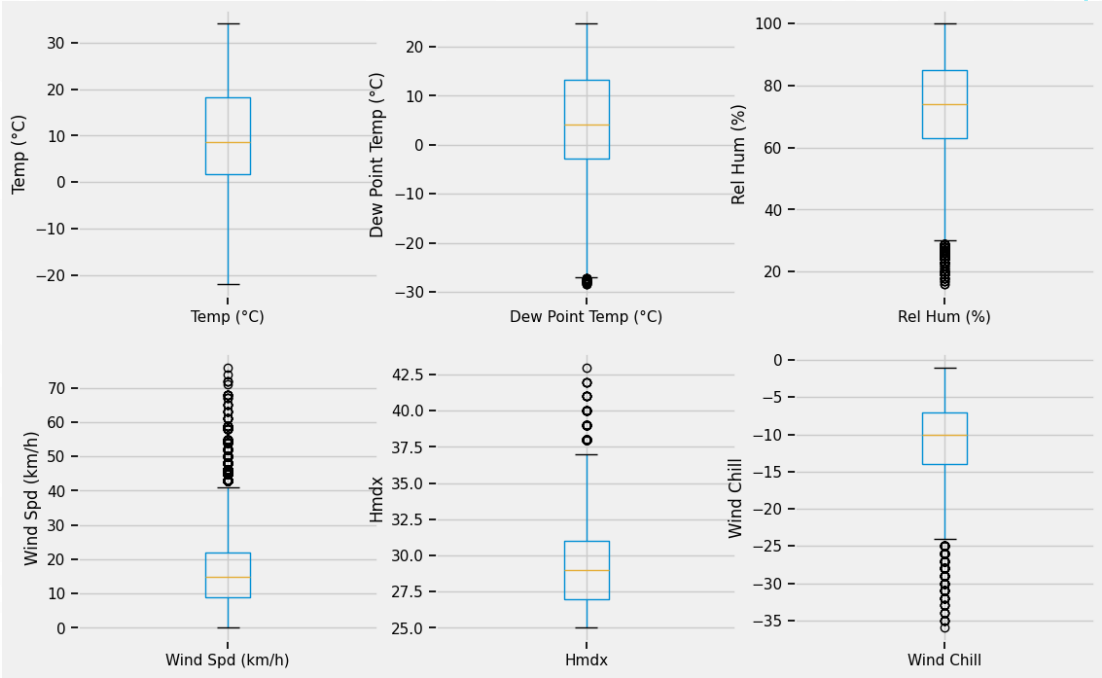
- Load Weather Data
- Rename the columns

Pre-processing

- Drop unwanted columns
- Check on duplicates
- Datetime Conversion

Clean

- Check for missing values
- **Check for outliers**



DATA CLEANING-RIDERSHIP DATA

Load	• **Load Ridership Data** • **Rename the columns**
Pre-processing	• Drop unwanted columns • Unify user type formats • Datetime Conversion • Duplicate trips
Clean	• Missing values and name inconsistency • Outliers
Merge	• Merge weather data with ridership data
Output	• Write merged dataframe to a csv file

```
[30]  1  ridership.info(show_counts=True)
      ✓ 7.4s                                                    Python
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15358993 entries, 0 to 15358992
Data columns (total 11 columns):
 #   Column              Non-Null Count     Dtype
---  ------              --------------     -----
 0   Trip Id             15358993 non-null  int64
 1   Start Time          15358993 non-null  object
 2   End Time            15358992 non-null  object
 3   Trip Duration       15358993 non-null  int64
 4   Start Station Id    14332100 non-null  float64
 5   Start Station Name  15262057 non-null  object
 6   End Station Id      14327681 non-null  float64
 7   End Station Name    15256259 non-null  object
 8   User Type           15358993 non-null  object
 9   Subscription Id     5052221 non-null   float64
 10  Bike Id             11943393 non-null  float64
dtypes: float64(4), int64(2), object(5)
memory usage: 1.3+ GB
```

DATA CLEANING-RIDERSHIP DATA

Load	• Load Ridership Data • Rename the columns
Pre-processing	• **Drop unwanted columns** • **Unify user type formats** • **Duplicate trips** • Datetime Conversion
Clean	• Missing values and name inconsistency • Outliers
Merge	• Merge weather data with ridership data
Output	• Write merged dataframe to a csv file

```
1  ridership['User Type'].unique()
```

✓ 0.7s

```
array(['Member', 'Casual', 'Annual Member', 'Casual Member'], dtype=object)
```

```
1  trip_duplicates = ridership[ridership.duplicated('Trip Id')]
2  trip_duplicates
```

[38] ✓ 1.3s Python

...

	Trip Id	Start Time	End Time	Trip Duration	Start Station Id	Start Station Name	End Station Id	End Station Name	User Type
9614632	10756846	03/02/2021 17:11:00 (EST)	03/02/2021 17:11:00 (EST)	0	7207.0	Dundas St W / Crawford St	7112.0	Liberty St / Fraser Ave Green P	Member
9735699	10909571	18/03/2021 17:32:00 (EST)	18/03/2021 17:32:00 (EST)	0	7351.0	Pretoria Av / Broadview Av	7286.0	Gerrard St E / Broadview - SMART	Member

DATA CLEANING-RIDERSHIP DATA

Load	• Load Ridership Data • Rename the columns
Pre-processing	• Drop unwanted columns • Unify user type formats • Duplicate trips • **Datetime Conversion**
Clean	• Missing values and name inconsistency • Outliers
Merge	• Merge weather data with ridership data
Output	• Write merged dataframe to a csv file

Split ridership data for different years

e.g.: ridership_2017

Datetime data time_convert

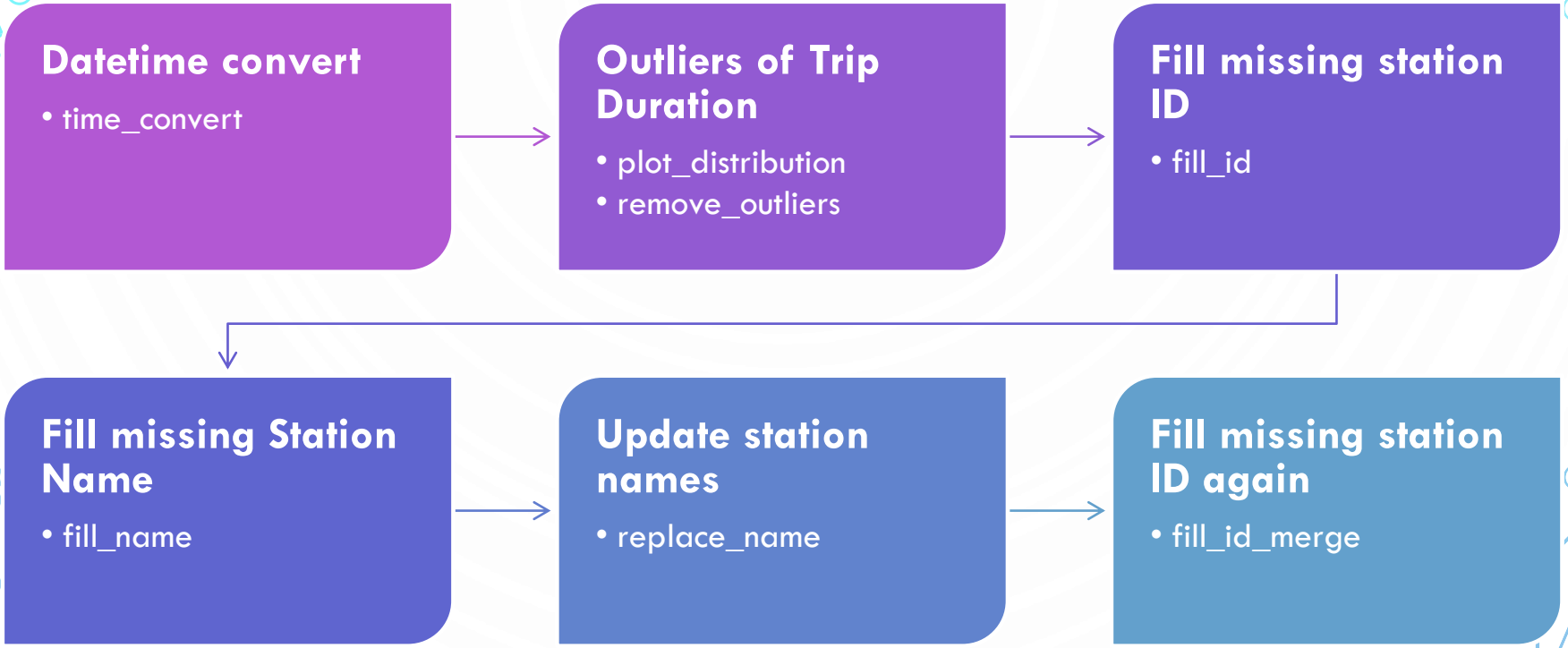
```python
# Create a time_convert function to convert Start Time and End Time into datetime variables
def time_convert(df):
    """Convert Start Time and End Time into datetime variables
    Using pd.to_datetime for each specific time format. DST is set to be True.
    """
    df_copy = df.copy()

    if 'UTC' in df_copy['Start Time'].iloc[0]:
        df_copy['Start Time'] = pd.to_datetime(df['Start Time'].str[:16], format='%Y-%m-%d %H:%M')\
            .dt.tz_localize('UTC', ambiguous=True, nonexistent='shift_backward')\
                .dt.tz_convert('US/Eastern')
        df_copy['End Time'] = pd.to_datetime(df['End Time'].str[:16], format='%Y-%m-%d %H:%M')\
            .dt.tz_localize('UTC', ambiguous=True, nonexistent='shift_backward')\
                .dt.tz_convert('US/Eastern')
    else:
        df_copy['Start Time'] = pd.to_datetime(df['Start Time'].str[:16], format='%d/%m/%Y %H:%M')\
            .dt.tz_localize('US/Eastern', ambiguous=True, nonexistent='shift_backward')
        df_copy['End Time'] = pd.to_datetime(df['End Time'].str[:16], format='%d/%m/%Y %H:%M')\
            .dt.tz_localize('US/Eastern', ambiguous=True, nonexistent='shift_backward')

    return df_copy
```

✓ 0.1s Python

DATA CLEANING-RIDERSHIP DATA

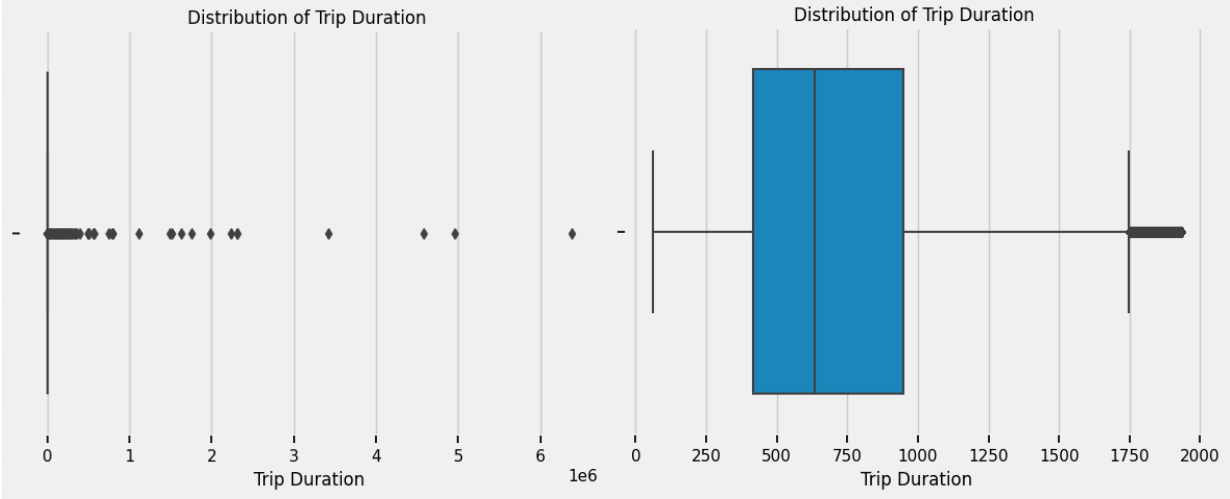


Ridership_est_2017 cleaning

```
[45]  ✓ 0.4s
```

```
# Remove trip duration outliers
ridership_est_2017 = remove_outliers(ridership_est_2017)
```

```
Percent of outliers removed: 7.65510161413967%
```

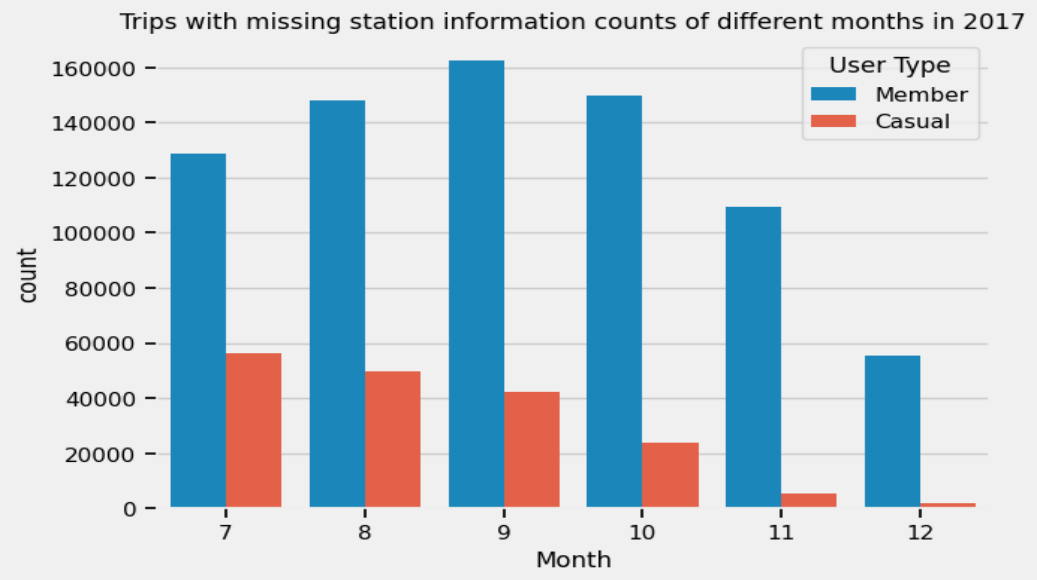


RIDERSHIP_EST_2017 CLEANING

```
1  ridership_est_2017.isnull().sum()
```

✓ 0.4s

```
Trip Id                 0
Start Time              0
End Time                0
Trip Duration           0
Start Station Id       934205
Start Station Name      0
End Station Id         934205
End Station Name        0
User Type               0
dtype: int64
```



RIDERSHIP_EST_2017 CLEANING

```python
# Create a fill_id function for filling missing station ID according to station name
def fill_id(df):
    """This function is for filling missing station names according to station ID
    using the bikeshare_stations data as reference.
    """

    df['Start Station Id'].fillna(
        df['Start Station Name'].map(bikeshare_stations.set_index('Station Name')['Station Id']),
        inplace=True)

    df['End Station Id'].fillna(
        df['End Station Name'].map(bikeshare_stations.set_index('Station Name')['Station Id']),
        inplace=True)

    return df
```

[49] ✓ 0.1s Python

```python
# Create a fill_name function for filling missing station name according to station ID
def fill_name(df):
    """This function is for filling missing station names according to station ID.
    using the bikeshare_stations as reference.
    """

    df['Start Station Name'].fillna(
        df['Start Station Id'].map(bikeshare_stations.set_index('Station Id')['Station Name']),
        inplace=True)

    df['End Station Name'].fillna(
        df['End Station Id'].map(bikeshare_stations.set_index('Station Id')['Station Name']),
        inplace=True)

    return df
```

[50] ✓ 0.9s Python

```python
ridership_est_2017.isnull().sum()
```

[47] ✓ 0.4s

```
Trip Id                0
Start Time             0
End Time               0
Trip Duration          0
Start Station Id       934205
Start Station Name     0
End Station Id         934205
End Station Name       0
User Type              0
dtype: int64
```

```python
fill_id(ridership_est_2017)
```

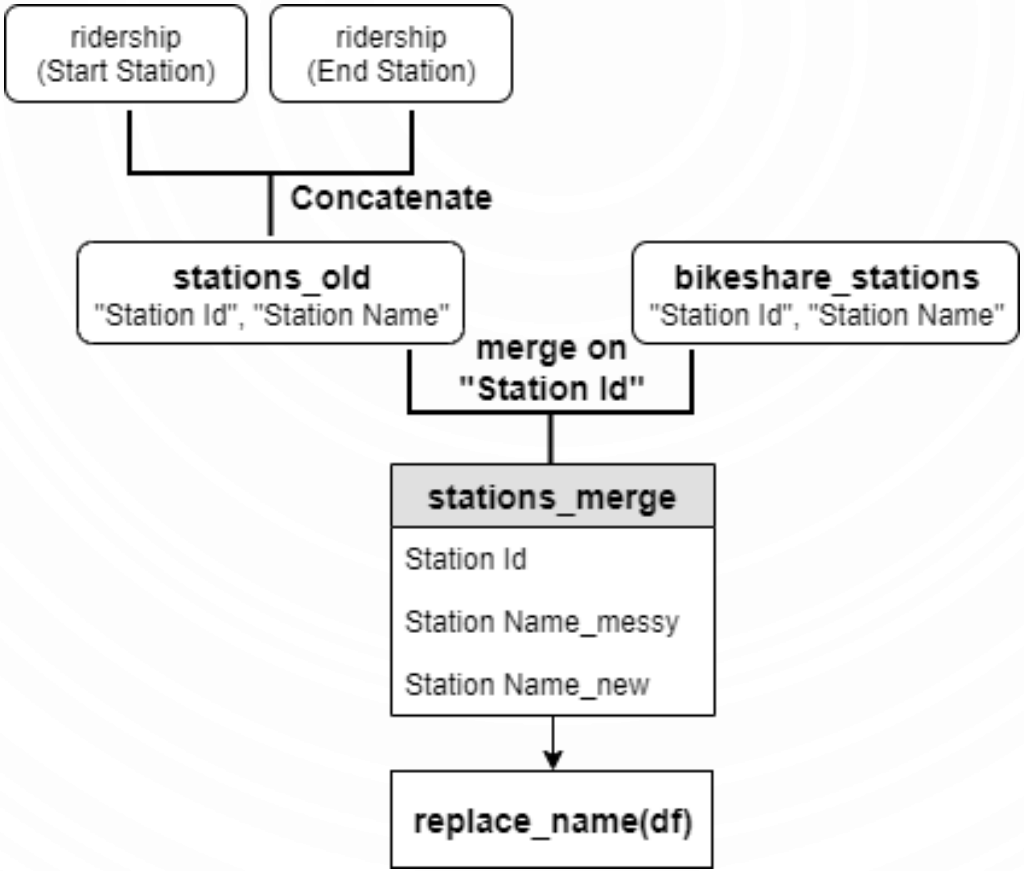
[51] ✓ 0.7s

```python
ridership_est_2017.isnull().sum()
```

[52] ✓ 0.4s

```
Trip Id                0
Start Time             0
End Time               0
Trip Duration          0
Start Station Id       236883
Start Station Name     0
End Station Id         241751
End Station Name       0
User Type              0
dtype: int64
```

RIDERSHIP_EST_2017 CLEANING



RIDERSHIP_EST_2017 CLEANING

Canceled stations: set new station names with _canceled

```
1  stations_merge['Station Name_new'].fillna(
2          stations_merge['Station Name_messy']+'_canceled',
3          inplace=True)
```

```
1  stations_merge[station_is_canceled]
```

✓ 0.1s Python

	Station Id	Station Name_messy	Station Name_new	lat	lon	capacity
16	7011.0	Wellington St W / Portland St	Wellington St W / Portland St_canceled	NaN	NaN	NaN
18	7013.0	Scott St / The Esplanade	Scott St / The Esplanade_canceled	NaN	NaN	NaN
22	7017.0	Widmer St / Adelaide St	Widmer St / Adelaide St _canceled	NaN	NaN	NaN
23	7017.0	Widmer St / Adelaide St W	Widmer St / Adelaide St W_canceled	NaN	NaN	NaN
25	7019.0	Temperance St / Yonge St	Temperance St / Yonge St_canceled	NaN	NaN	NaN
...	...	...	...	...	...	...
762	7638.0	Warehouse EBS-Station	Warehouse EBS-Station_canceled	NaN	NaN	NaN
777	7649.0	Greenwood Subway Station	Greenwood Subway Station_canceled	NaN	NaN	NaN
778	7649.0	Greenwood Subway Station - SMART	Greenwood Subway Station - SMART_canceled	NaN	NaN	NaN
780	7651.0	Bloor St W / Gladstone Ave	Bloor St W / Gladstone Ave_canceled	NaN	NaN	NaN
781	7653.0	Bloor St W / Indian Rd	Bloor St W / Indian Rd_canceled	NaN	NaN	NaN

64 rows × 6 columns

RIDERSHIP_EST_2017 CLEANING

```
1  ridership_est_2017[ridership_est_2017['Start Station Name']=='Beverley St / College St']
```

✓ 0.5s Python

	Trip Id	Start Time	End Time	Trip Duration	Start Station Id	Start Station Name	End Station Id	End Station Name	User Type
458	712918	2017-01-01 20:41:00-05:00	2017-01-01 20:51:00-05:00	598	7161.0	Beverley St / College St	7026.0	Bay St / St. Joseph St	Member
459	712917	2017-01-01 20:41:00-05:00	2017-01-01 [illegible] 05:00		7161.0	Beverley St / College St	7026.0	Bay St / St. Joseph St	Member
529	712988	2017-01-02 02:48:00-05:00	2017-01-02 02:59:00-05:00	680	7161.0	Beverley St / College St	7121.0	Jarvis St / Dundas St E	Member
795	713334	2017-01-02 14:15:00-05:00	2017-01-02 14:19:00-05:00	235	7161.0	Beverley St / College St	7006.0	Bay St / College St (East Side)	Member
1026	713643	2017-01-02 17:26:00-05:00	2017-01-02 17:33:00-05:00	412	7161.0	Beverley St / College St	7057.0	Simcoe St / Wellington St South	Member
...	...	...	...	...	...	...	...	...	...
1491881	1971195	2017-09-30 23:17:00-04:00	2017-09-30 23:20:00-04:00	223	NaN	Beverley St / College St	7078.0	College St / Major St	Member
1492024	1971340	2017-09-30 23:48:00-04:00	2017-09-30 23:56:00-04:00	495	NaN	Beverley St / College St	7260.0	Spadina Ave / Adelaide St W	Member
1492049	1971365	2017-09-30 23:55:00-04:00	2017-10-01 00:01:00-04:00	406	NaN	Beverley St / College St	7032.0	Augusta Ave / Dundas St W	Member
1492118	1971434	2017-10-01 00:06:00-04:00	2017-10-01 00:32:00-04:00	1567	NaN	Beverley St / College St	7045.0	Bond St / Queen St E	Member

Ridership_est_2017 cleaning

RIDERSHIP_EST_2017 CLEANING

Create another function fill_id_merge to fill missing ID using the stations_merge df

```
1 ridership_est_2017.isnull().sum()
```
[47] ✓ 0.4s

Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	934205
Start Station Name	0
End Station Id	934205
End Station Name	0
User Type	0

dtype: int64

```
1 fill_id(ridership_est_2017)
```
[51] ✓ 0.7s

```
1 ridership_est_2017.isnull().sum()
```
[52] ✓ 0.4s

Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	236883
Start Station Name	0
End Station Id	241751
End Station Name	0
User Type	0

dtype: int64

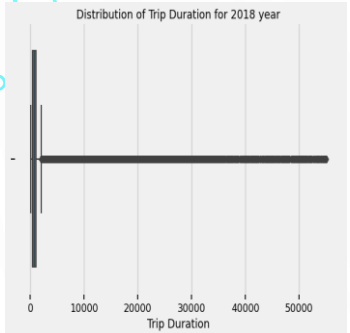
```
1 # fill missing ID with new station names
2 fill_id_merge(ridership_est_2017)
```
✓ 0.5s

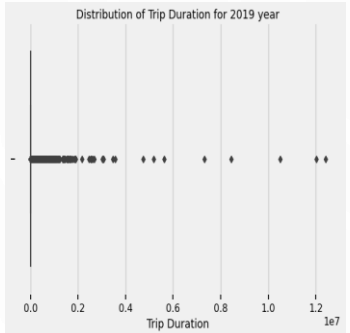
```
1 ridership_est_2017.isnull().sum()
```
✓ 0.4s

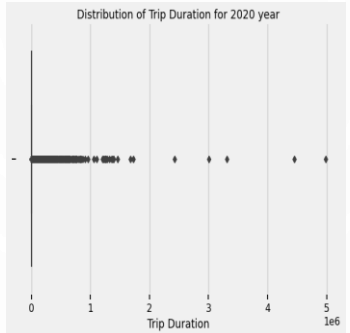
Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	15279
rt Station Name	0
End Station Id	17685
End Station Name	0
User Type	0

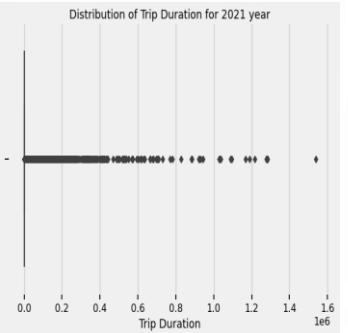
dtype: int64

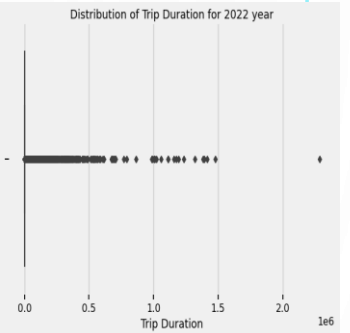
DATA CLEANING FOR RIDERSHIP DATA 2018 TO 2022

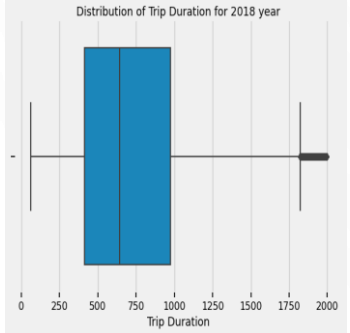


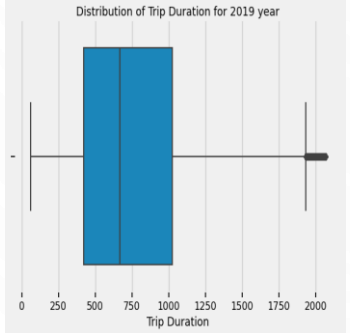


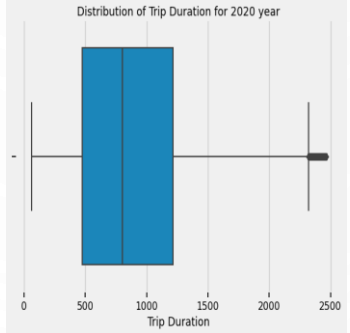


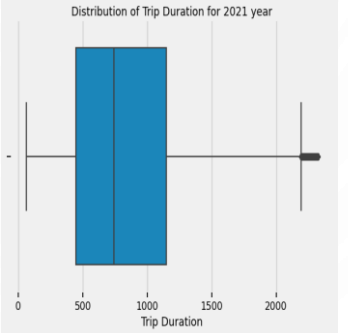


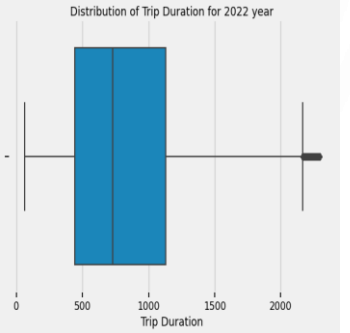













```
Plot_distribution -> remove_outliers
```

DATA CLEANING FOR RIDERSHIP DATA 2018 TO 2022

(fill_id) -> (fill_name) -> replace_name -> fill_id_merge

```
1 ridership_est_2018.isnull().sum()
✓ 0.6s
```

Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	56
Start Station Name	0
End Station Id	56
End Station Name	0
User Type	0

dtype: int64

```
1 ridership_est_2018.isnull().sum()
✓ 0.5s
```

Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	18
Start Station Name	0
End Station Id	13
End Station Name	0
User Type	0

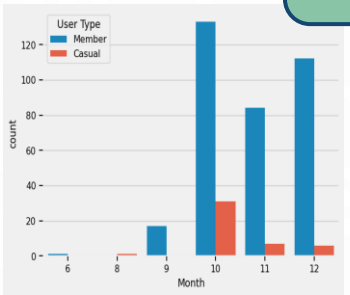
dtype: int64

2018

```
1 ridership_est_2019.isnull().sum()
✓ 0.7s
```

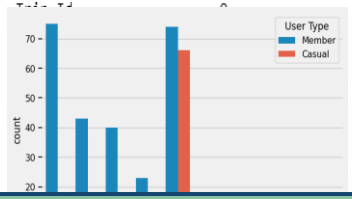
Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	0
Start Station Name	0
End Station Id	392
End Station Name	392
User Type	0

dtype: int64



2019

```
1 ridership_est_2020.isnull().sum()
✓ 0.8s
```



Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	0
Start Station Name	0
End Station Id	396
End Station Name	396
User Type	0

dtype: int64

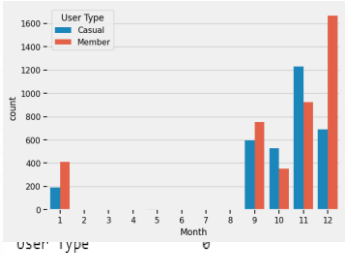
2020

```
1 ridership_est_2021.isnull().sum()
✓ 1.7s
```

Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	0
Start Station Name	3596
End Station Id	47
tation Name	3886
Type	0

: int64

```
1 ridership_est_2021.isnull().sum()
✓ 1.7s
```



dtype: int64

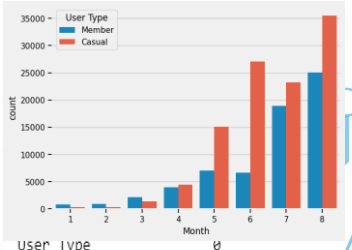
2021

```
1 ridership_est_2022.isnull().sum()
✓ 0.9s
```

Trip Id	0
Start Time	0
End Time	0
Trip Duration	0
Start Station Id	0
Start Station Name	88783
End Station Id	14
End Station Name	89757
User Type	0

dtype: int64

```
1 ridership_est_2022.isnull().sum()
✓ 0.9s
```



dtype: int64

2022

Clean ridership data: “ridership_clean”

DATA CLEANING-RIDERSHIP DATA

Load	•Load Ridership Data •Rename the columns
Pre-processing	•Drop unwanted columns •Unify user type formats •Datetime Conversion •Duplicate trips
Clean	•Missing values and name inconsistency •Outliers •Check for feature distribution
Merge	•**Merge weather data with ridership data**
Output	•**Write merged dataframe to a csv file**

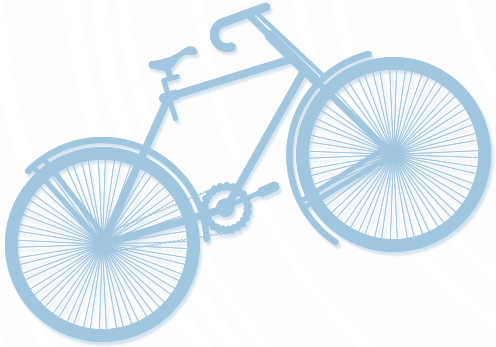
```
1  df_merged.info(show_counts=True)
```

✓ 9.3s

Python

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 14630836 entries, 0 to 15358706
Data columns (total 20 columns):
 #   Column                Non-Null Count     Dtype
---  ------                --------------     -----
 0   Trip Id               14630836 non-null  int64
 1   Start Time            14630836 non-null  datetime64[ns, US/Eastern]
 2   End Time              14630836 non-null  datetime64[ns, US/Eastern]
 3   Trip Duration         14630836 non-null
 4   Start Station Id      14630836 non-null
 5   Start Station Name    14630836 non-null
 6   End Station Id        14630836 non-null
 7   End Station Name      14630836 non-null
 8   User Type             14630836 non-null
 9   Start Hour            14630836 non-null
 10  Temp (°C)             14630836 non-null
 11  Dew Point Temp (°C)   14630836 non-null  float64
 12  Rel Hum (%)           14630836 non-null  float64
 13  Wind Dir (10s deg)    14630836 non-null  float64
 14  Wind Spd (km/h)       14630836 non-null  float64
 15  Visibility (km)       14630836 non-null  float64
 16  Hmdx                  4989189 non-null   float64
 17  Wind Chill            695126 non-null    float64
 18  Weather               14630836 non-null  object
 19  Precip. Amount (mm)   14630836 non-null  float64
dtypes: datetime64[ns, US/Eastern](3), float64(11), int64(2), object(4)
memory usage: 2.3+ GB
```

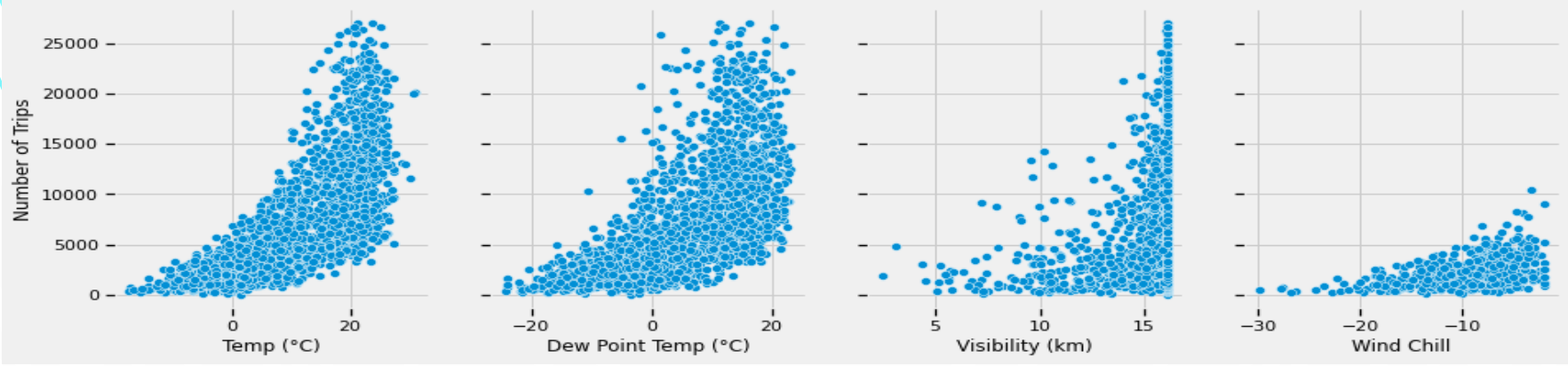
Merged data:
"df_merged",
"df_merged_stations"

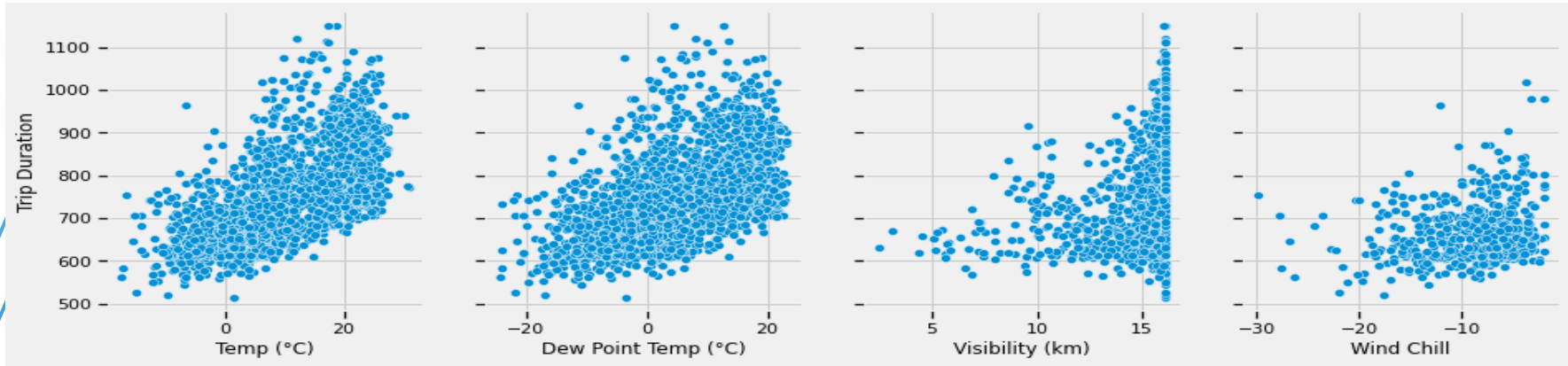


EXPLORATORY DATA ANALYSIS

Temporal

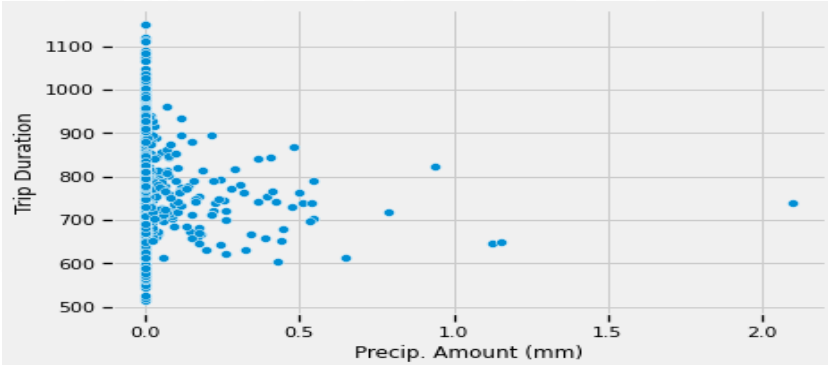
TEMPORAL EDA :CLIMATE

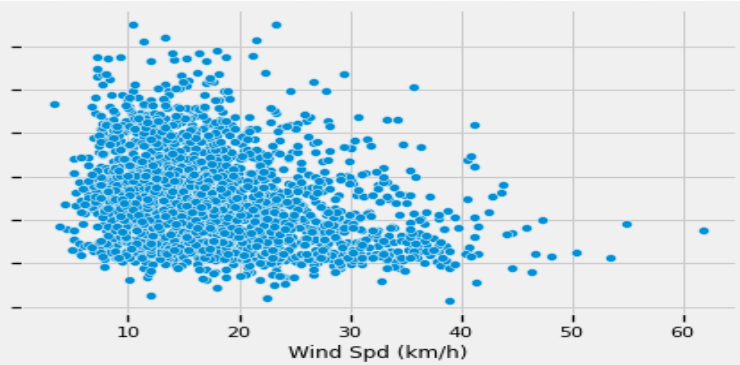


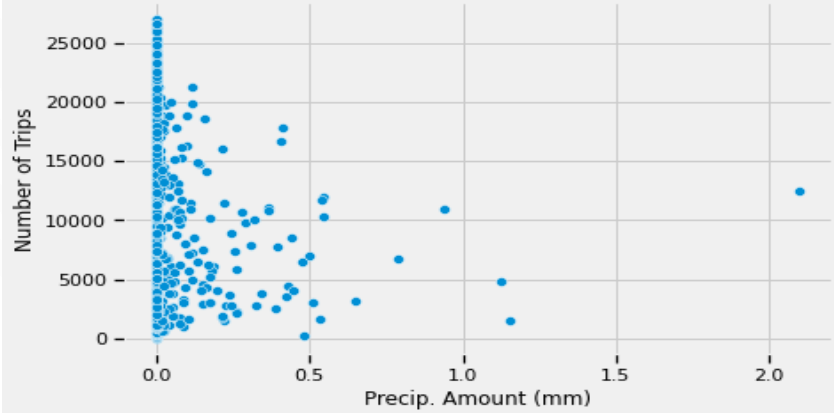


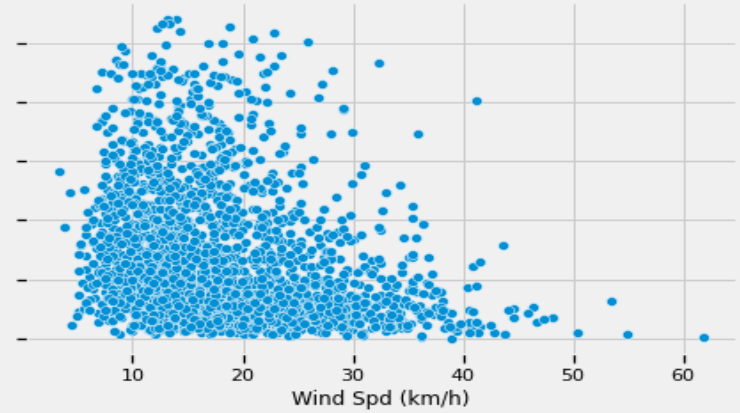
Features with Positive Correlation in terms of Number of Trips and Trip Duration

TEMPORAL EDA :CLIMATE



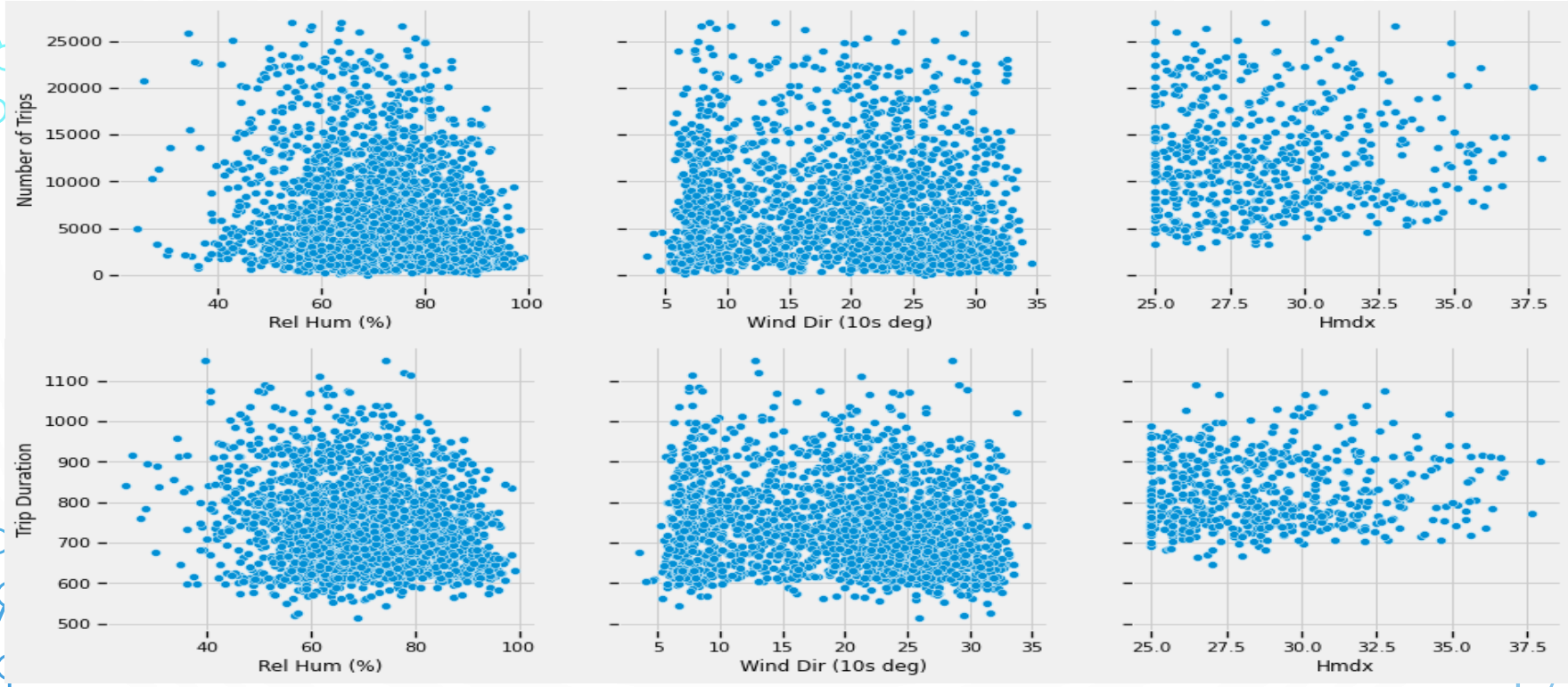






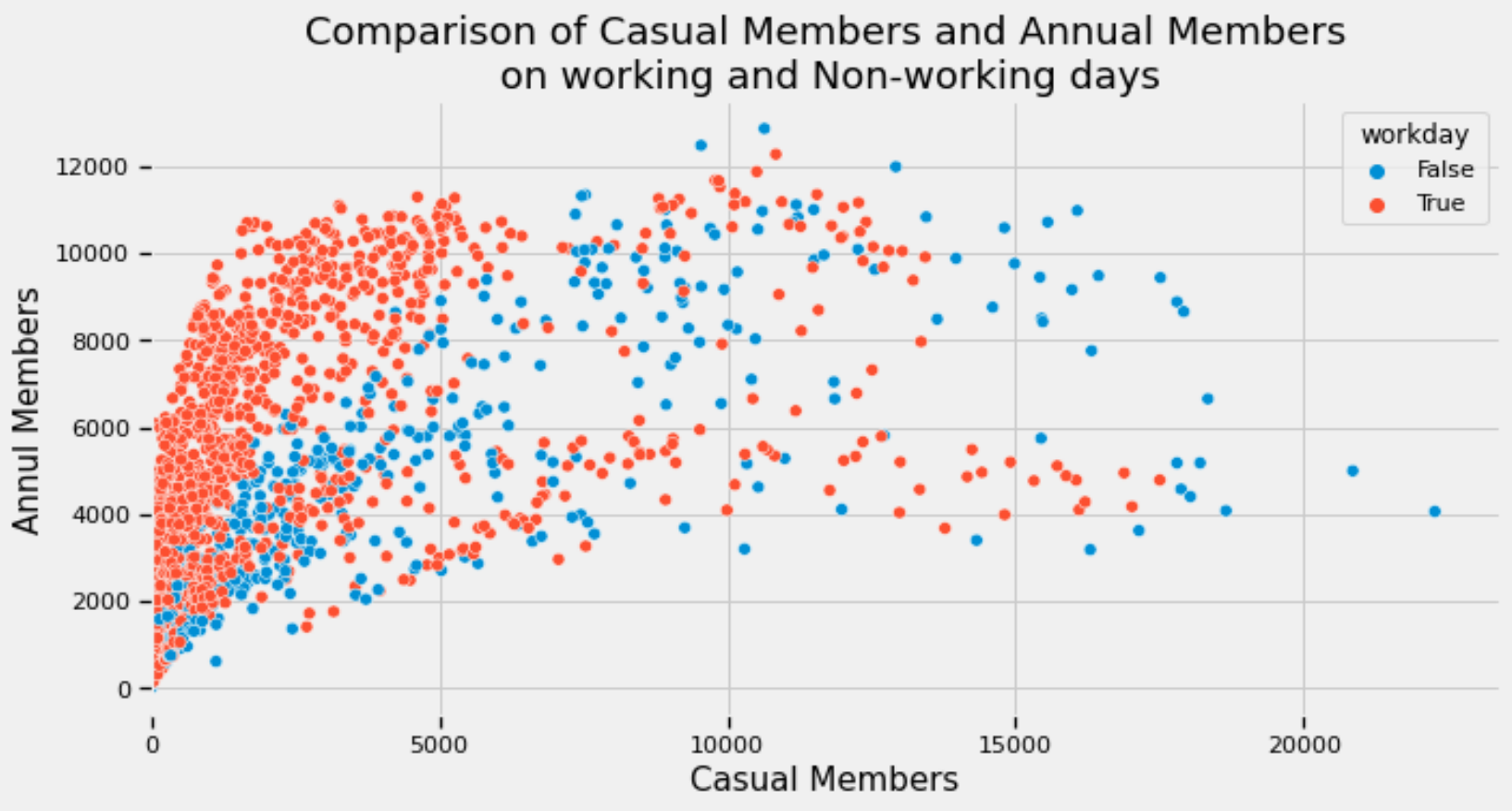
Features with Negative Correlation in terms of Number of Trips and Trip Duration

TEMPORAL EDA :CLIMATE

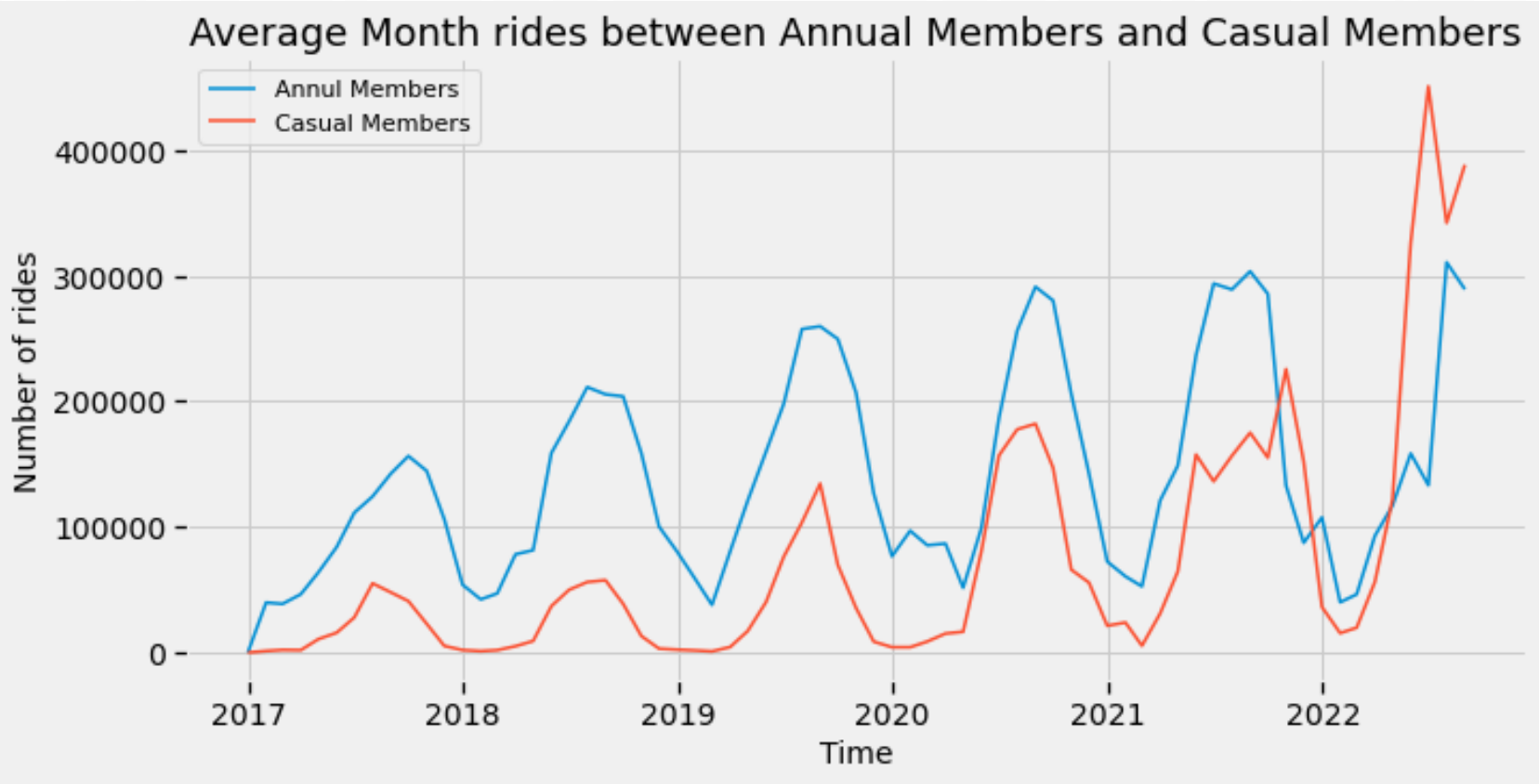


Features with No Correlation in terms of Number of Trips and Trip Duration

TEMPORAL EDA :WORKDAYS VS WEEKEND

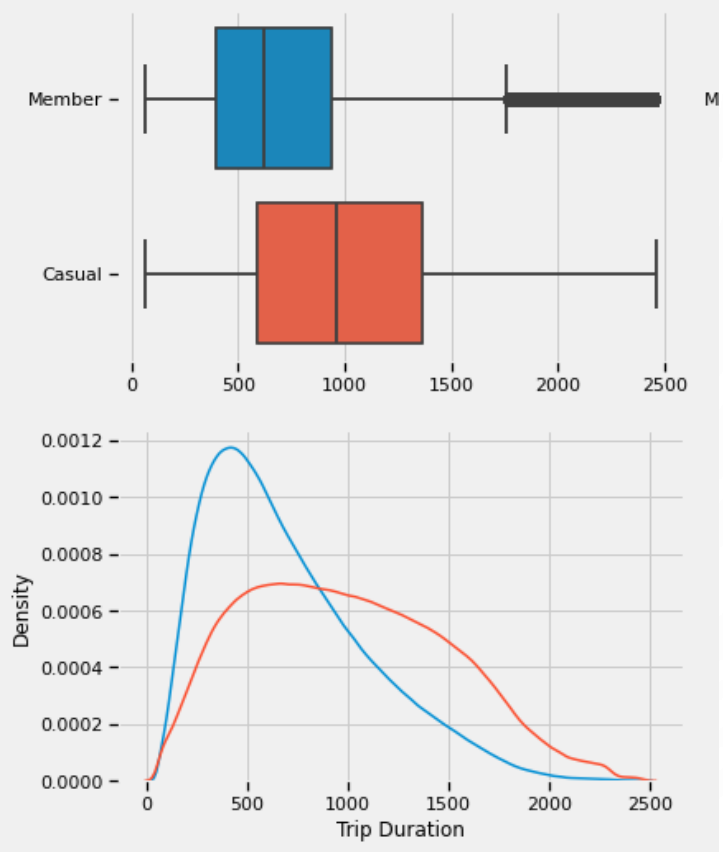


TEMPORAL EDA :MONTH IN YEAR

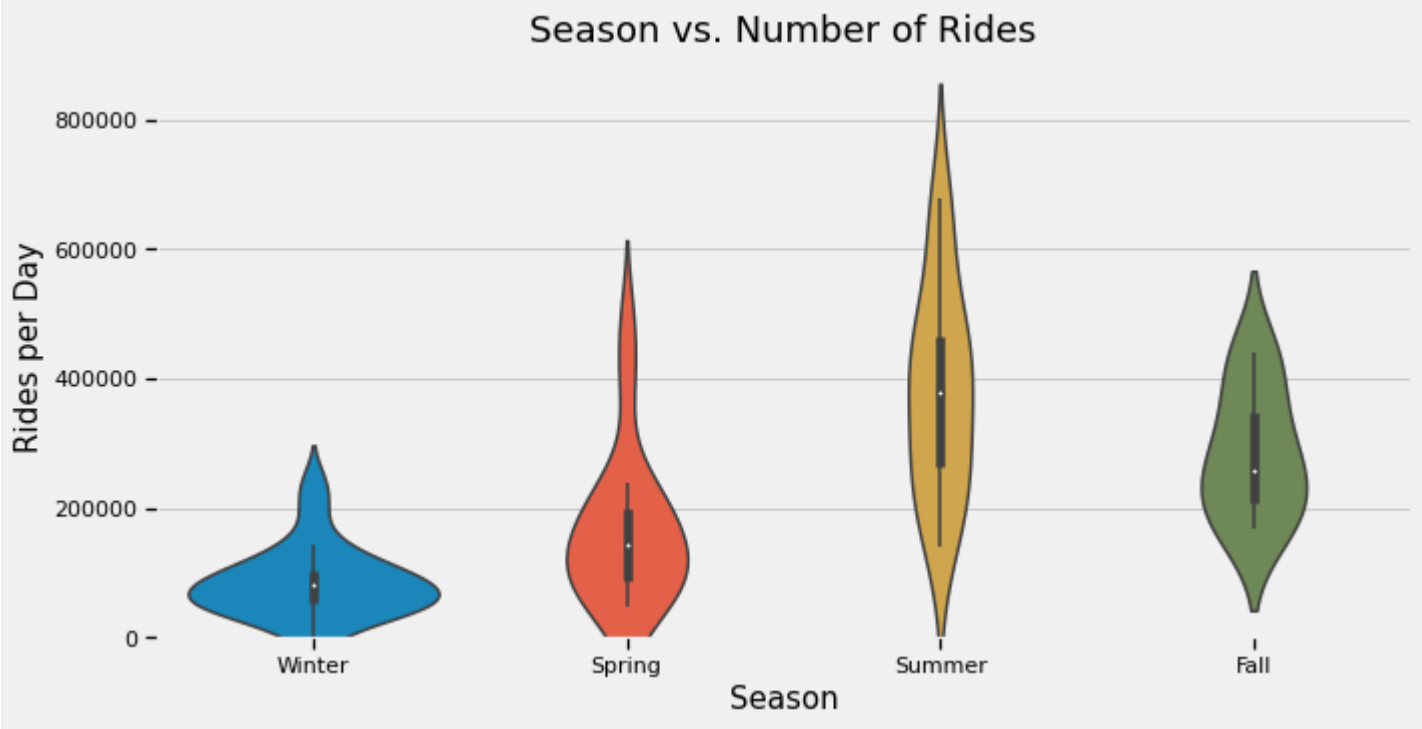


TEMPORAL EDA :MONTHLY DISTRIBUTION

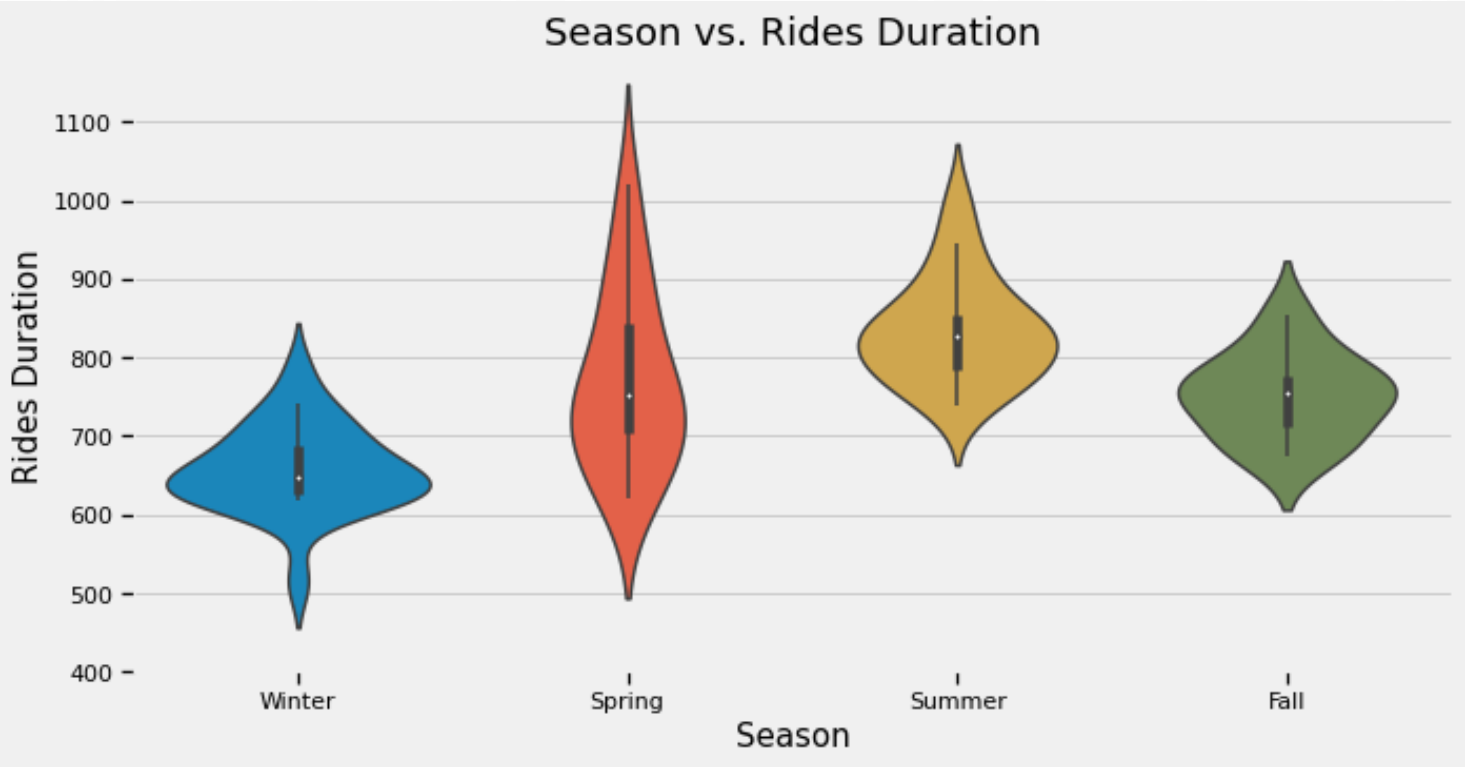
The Trips Duration between Casual Members and Annual Members on Monthly Basis



TEMPORAL EDA :SEASONAL



TEMPORAL EDA :SEASONAL



TOP 5 MOST POPULAR ROUTES

1. Ontario Place Blvd / Lakeshore Blvd W - Ontario Place Blvd / Lakeshore Blvd W

2. Tommy Thompson Park (Leslie Street Spit) - Tommy Thompson Park (Leslie Street Spit)

3. HTO Park (Queens Quay W) - HTO Park (Queens Quay W)

4. Cherry Beach - Tommy Thompson Park (Leslie Street Spit)

5. Bathurst St/Queens Quay(Billy Bishop Airport)- York St / Queens Quay W

THE TOP 5 MOST POPULAR ROUTE

SUMMER

1. Ontario Place Blvd / Lakeshore Blvd W - Ontario Place Blvd / Lakeshore Blvd W
2. Tommy Thompson Park (Leslie Street Spit) - Tommy Thompson Park (Leslie Street Spit)
3. HTO Park (Queens Quay W) - HTO Park (Queens Quay W)
4. Cherry Beach - Tommy Thompson Park (Leslie Street Spit)
5. Humber Bay Shores Park West - Humber Bay Shores Park West

WINTER

1. Front St W / Blue Jays Way - Union Station
2. Bathurst St/Queens Quay(Billy Bishop Airport) - York St / Queens Quay W
3. York St / Queens Quay W - Bathurst St/Queens Quay(Billy Bishop Airport)
4. Cariboo St / Rail Path - Bloor GO / UP Station/ Rail Path_canceled
5. St. George St / Bloor St W - St. George St / Russell St - SMART

THE TOP 5 MOST POPULAR ROUTE

FALL

1. Ontario Place Blvd / Lakeshore Blvd W - Ontario Place Blvd / Lakeshore Blvd W
2. Tommy Thompson Park (Leslie Street Spit) - Tommy Thompson Park (Leslie Street Spit)
3. Front St W / Blue Jays Way - Union Station
4. Bathurst St/Queens Quay(Billy Bishop Airport) - York St / Queens Quay W
5. York St / Queens Quay W – Bathurst St/Queens Quay(Billy Bishop Airport)

SPRING

1. Tommy Thompson Park (Leslie Street Spit) - Tommy Thompson Park (Leslie Street Spit)
2. Ontario Place Blvd / Lakeshore Blvd W - Ontario Place Blvd / Lakeshore Blvd W
3. Cherry Beach - Tommy Thompson Park (Leslie Street Spit)
4. Humber Bay Shores Park West - Humber Bay Shores Park West
5. HTO Park (Queens Quay W) - HTO Park (Queens Quay W)

THE TOP 5 MOST POPULAR ROUTE

ANNUAL MEMBERS

1. Front St W / Blue Jays Way - Union Station
2. Bathurst St/Queens Quay(Billy Bishop Airport) - York St / Queens Quay W
3. York St / Queens Quay W - Bathurst St/Queens Quay(Billy Bishop Airport)
4. Tommy Thompson Park (Leslie Street Spit) - Tommy Thompson Park (Leslie Street Spit)
5. Wellesley St E / Yonge St Green P_canceled - Sherbourne St / Wellesley St E

WINTER

1. Front St W / Blue Jays Way - Union Station
2. Bathurst St/Queens Quay(Billy Bishop Airport) - York St / Queens Quay W
3. York St / Queens Quay W - Bathurst St/Queens Quay(Billy Bishop Airport)
4. Cariboo St / Rail Path - Bloor GO / UP Station/ Rail Path_canceled
5. St. George St / Bloor St W - St. George St / Russell St - SMART

THE TOP 5 MOST POPULAR ROUTE

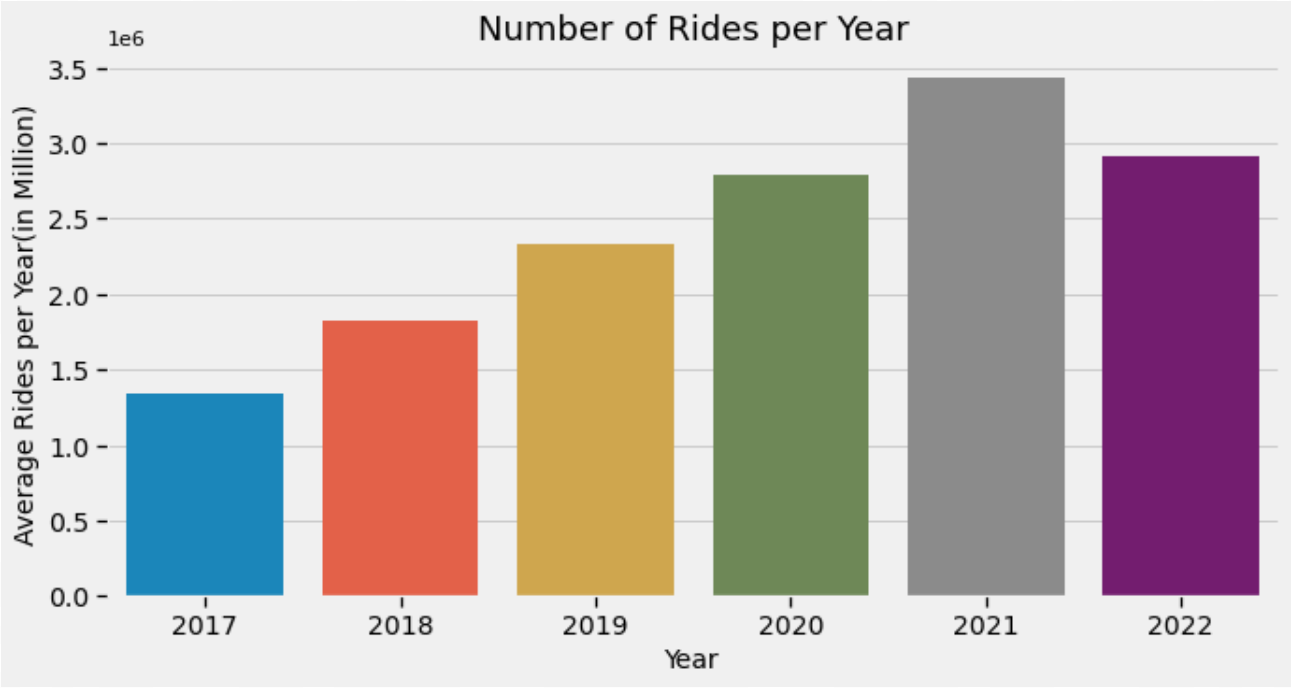
CASUAL MEMBERS

- **1. Ontario Place Blvd / Lakeshore Blvd W - Ontario Place Blvd / Lakeshore Blvd W**
- **2. Tommy Thompson Park (Leslie Street Spit) - Tommy Thompson Park (Leslie Street Spit)**
- **3. HTO Park (Queens Quay W) - HTO Park (Queens Quay W)**
- **4. Humber Bay Shores Park West - Humber Bay Shores Park West**
- **5. Lakeshore Blvd W / Ontario Dr - Lakeshore Blvd W / Ontario Dr**

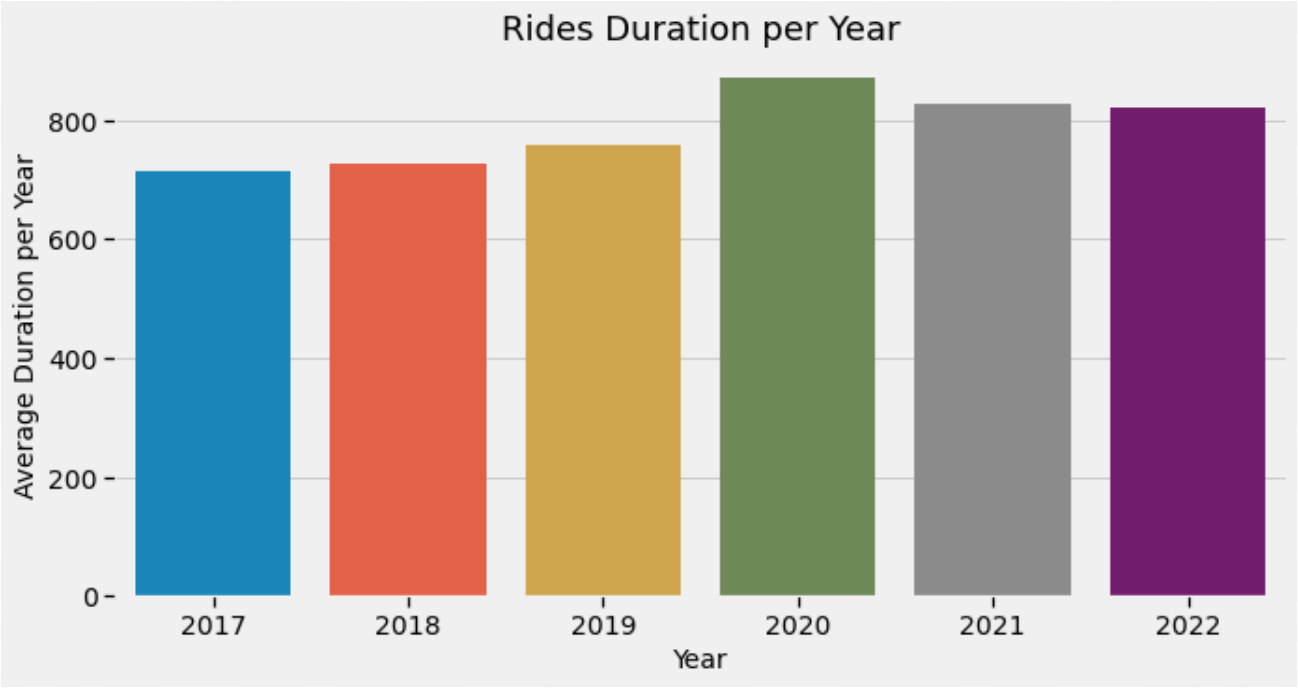
SUMMER

- **1. Ontario Place Blvd / Lakeshore Blvd W - Ontario Place Blvd / Lakeshore Blvd W**
- **2. Tommy Thompson Park (Leslie Street Spit) - Tommy Thompson Park (Leslie Street Spit)**
- **3. HTO Park (Queens Quay W) - HTO Park (Queens Quay W)**
- **4. Cherry Beach - Tommy Thompson Park (Leslie Street Spit)**
- **5. Humber Bay Shores Park West - Humber Bay Shores Park West**

TEMPORAL EDA: ANNUAL



TEMPORAL EDA: ANNUAL



TOP 5 BUSIEST START STATIONS

BEFORE PANDEMIC

1. Union Station
2. Bay St / Wellesley St W
3. York St / Queens Quay W
4. Simcoe St / Wellington St South
5. Dundas St W / Yonge St

DURING PANDEMIC

1. York St / Queens Quay W
2. Lake Shore Blvd W / Ontario Dr
3. Queens Quay E / Lower Sherbourne St
4. Bay St / Queens Quay W (Ferry Terminal)
5. Ontario Place Blvd / Lake Shore Blvd W (East)

GEOSPATIAL EDA :CONCLUSION

When?

Casual/Daily

Monthly/Yearly

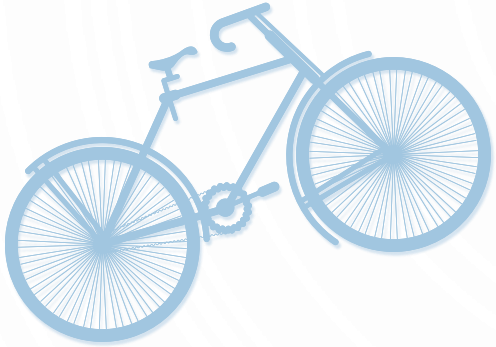
Summer

Spring

Fall

Spring

Summer



EXPLORATORY DATA ANALYSIS

Geospatial

EXPLORATORY DATA ANALYSIS

Neighbourhoods

Which ones has most and longest rides?

Stations

Top 10 stations with rides for diff users

Subway Stations

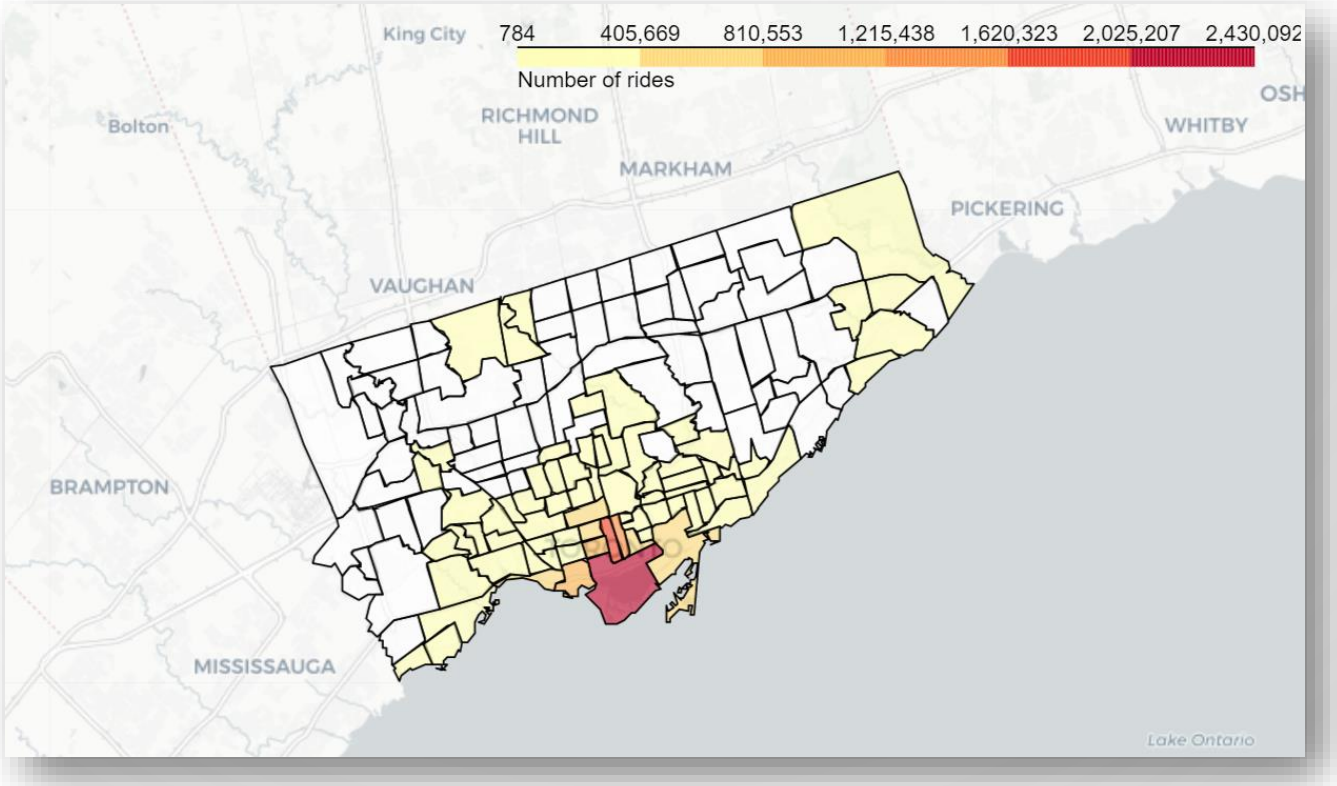
Identify the subway stations close to the top bike stations

Bike Lanes

For future work ! We want to explore the congested lanes

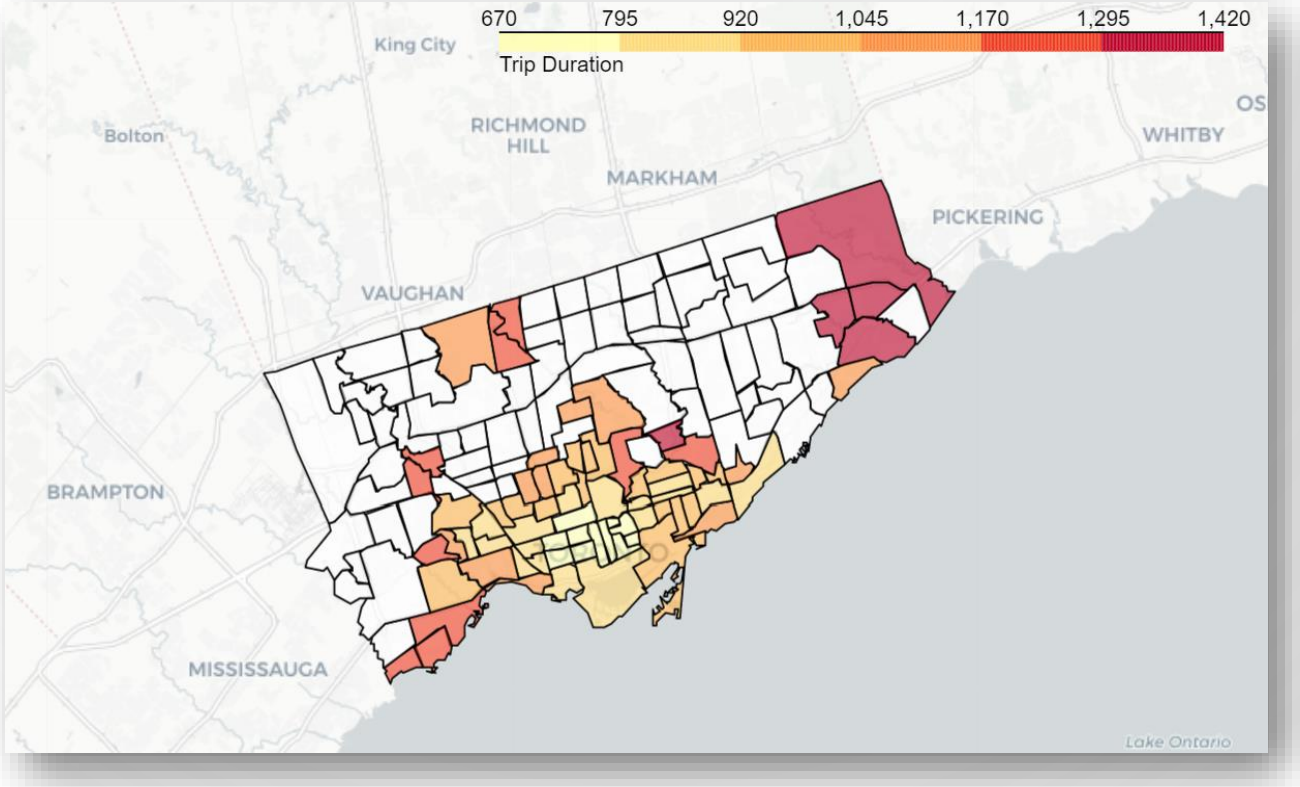
GEOSPATIAL EDA :NEIGHBOURHOODS

How many rides start/end?



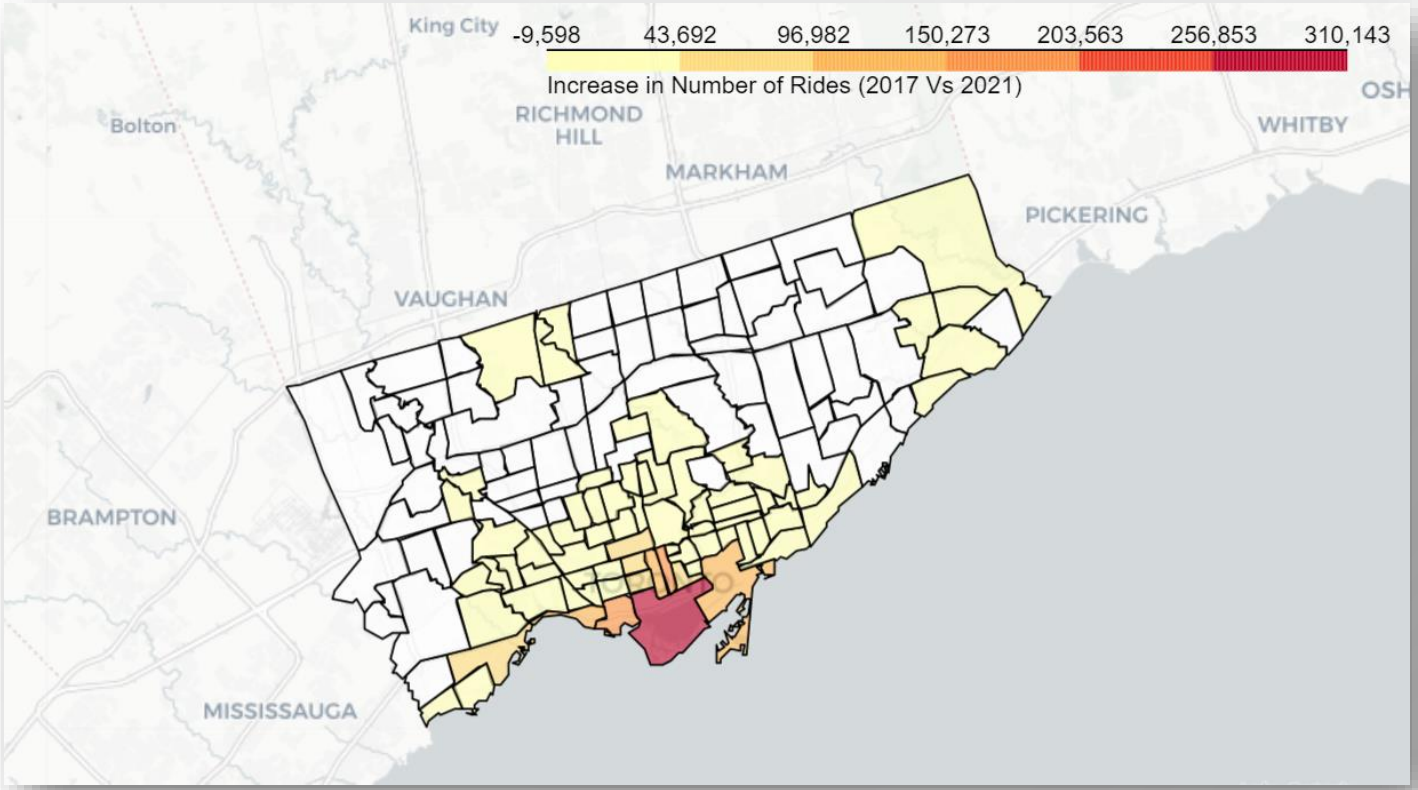
GEOSPATIAL EDA :NEIGHBOURHOODS

What is the average trip duration?



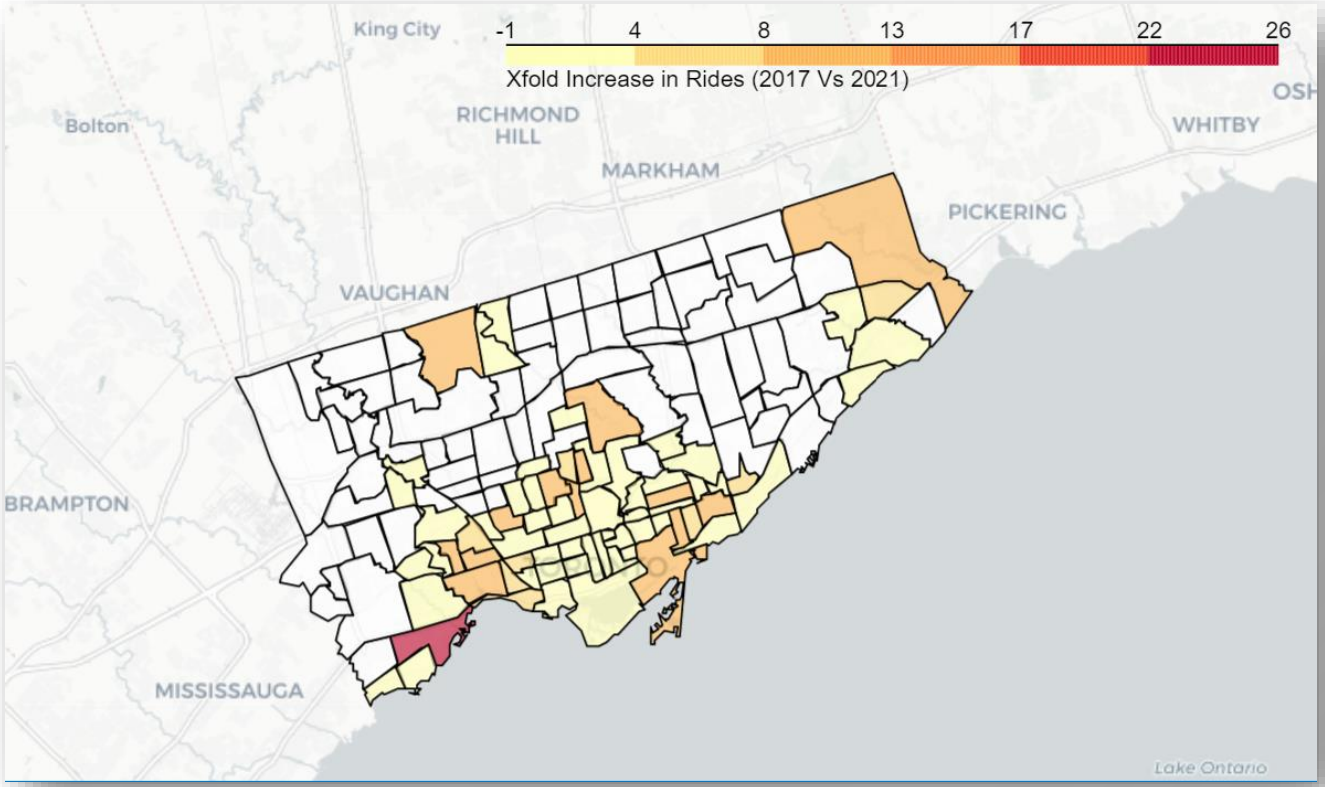
GEOSPATIAL EDA :NEIGHBOURHOODS

How many rides increased from 2017 to 2021?



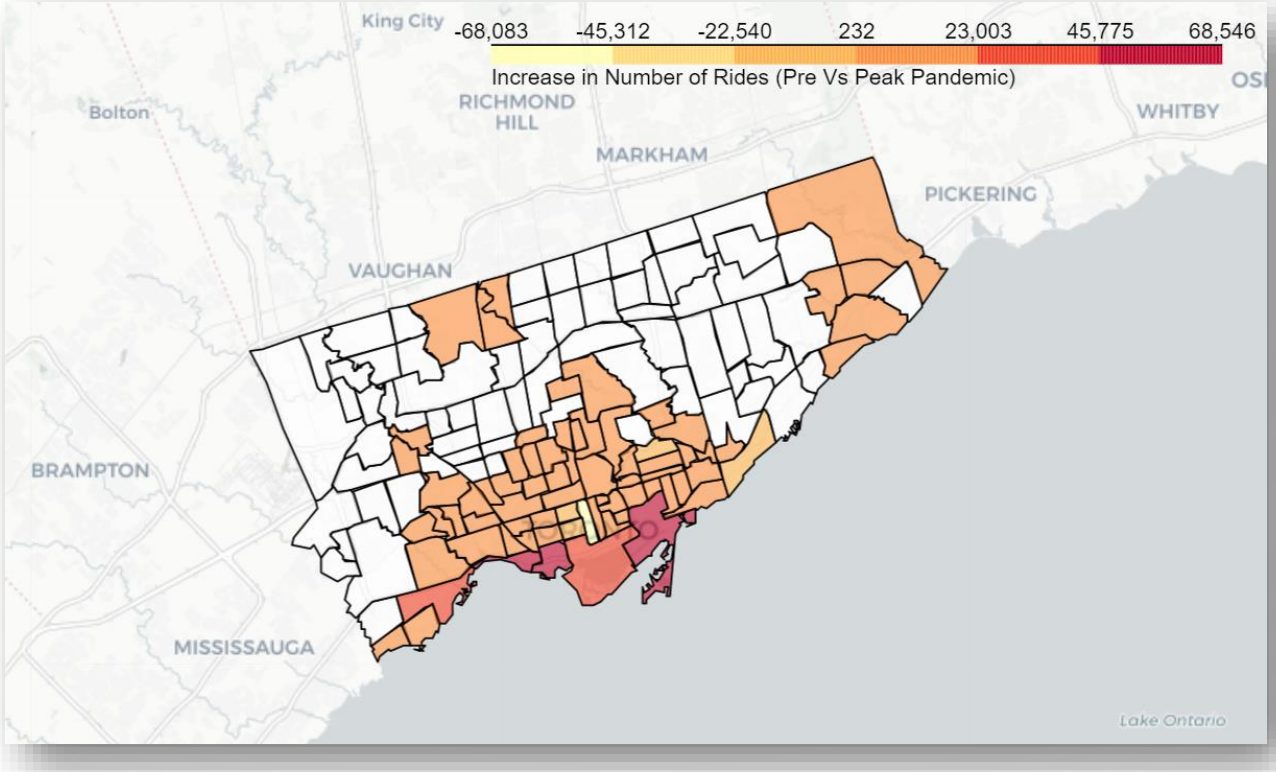
GEOSPATIAL EDA :NEIGHBOURHOODS

But, how much did rides increase when normalized?



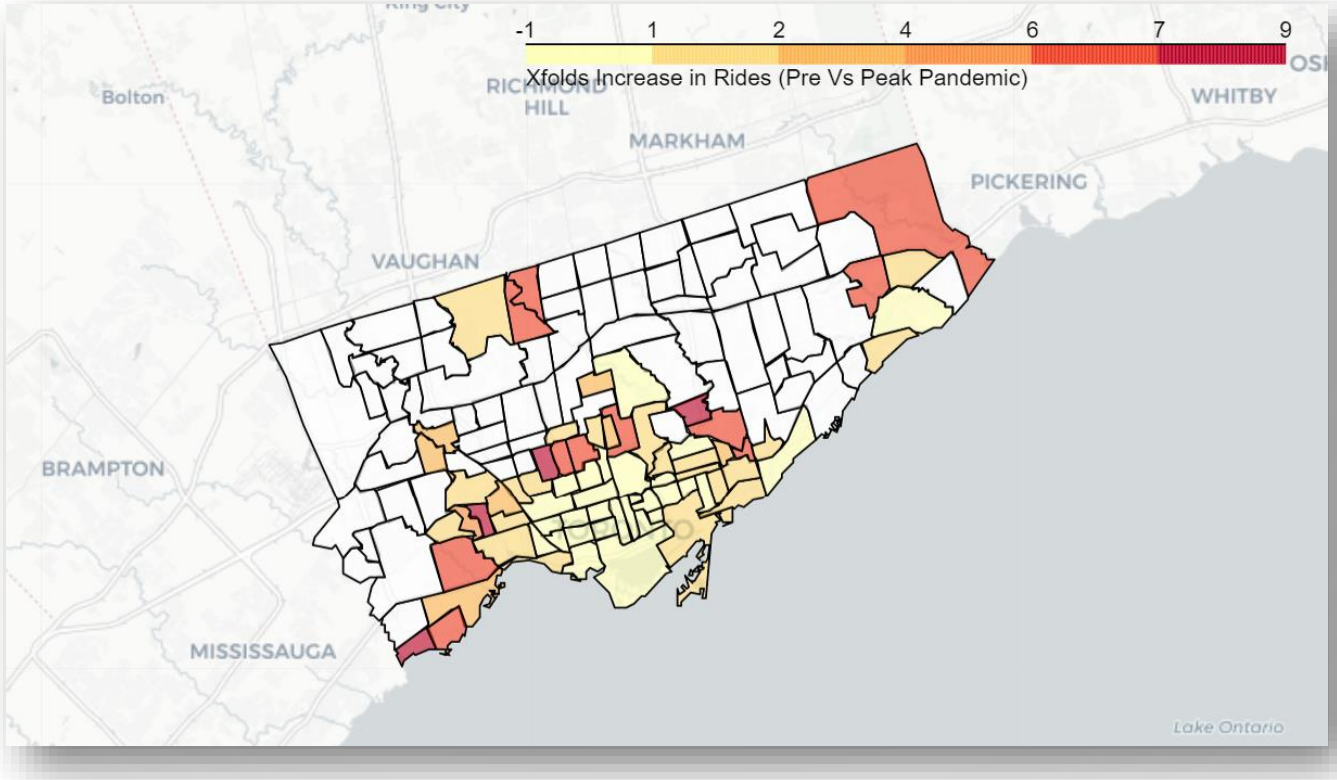
GEOSPATIAL EDA :NEIGHBOURHOODS

What happened to ride numbers due to the Pandemic?



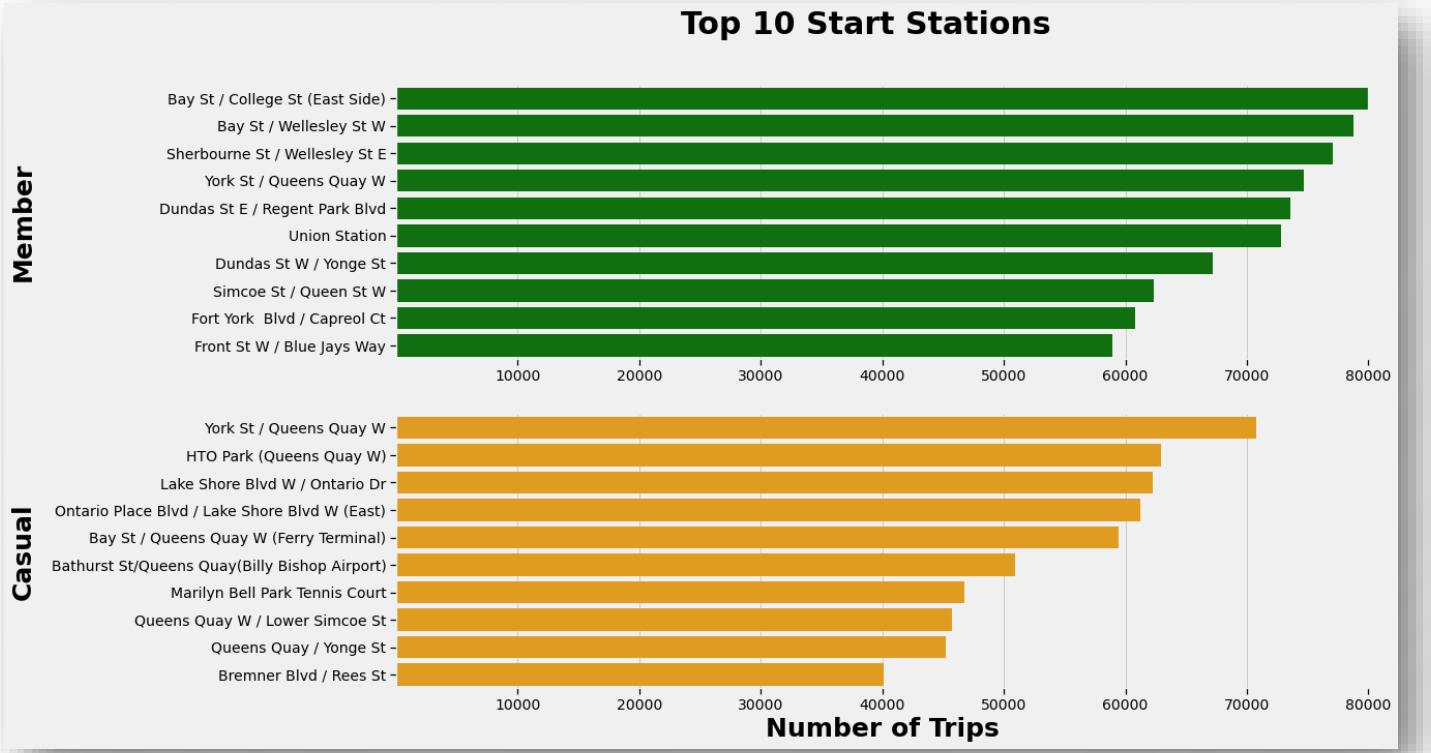
GEOSPATIAL EDA :NEIGHBOURHOODS

What happened to ride numbers due the Pandemic, normalized?



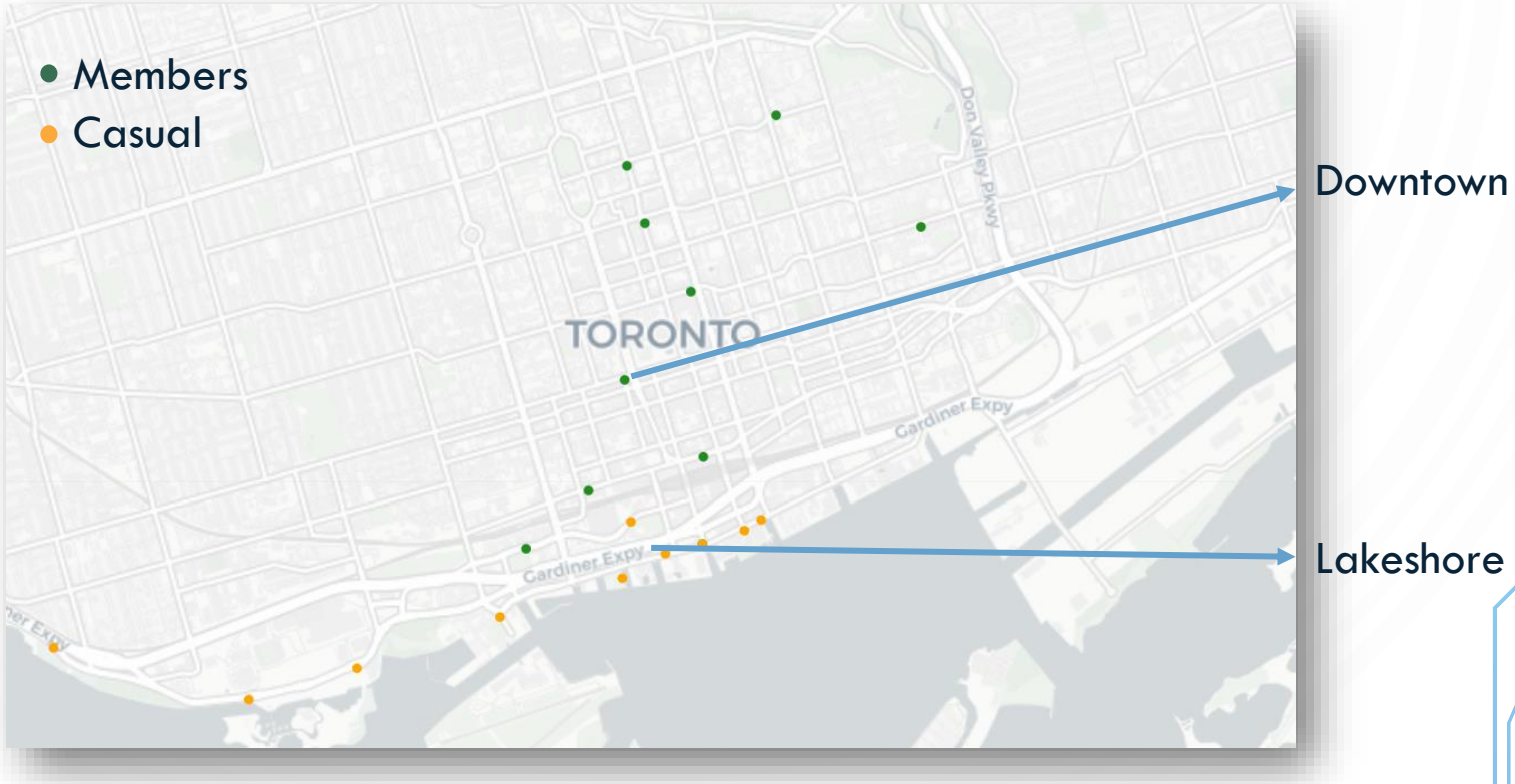
GEOSPATIAL EDA :STATIONS

What are the top 10 start stations? Members Vs Casual



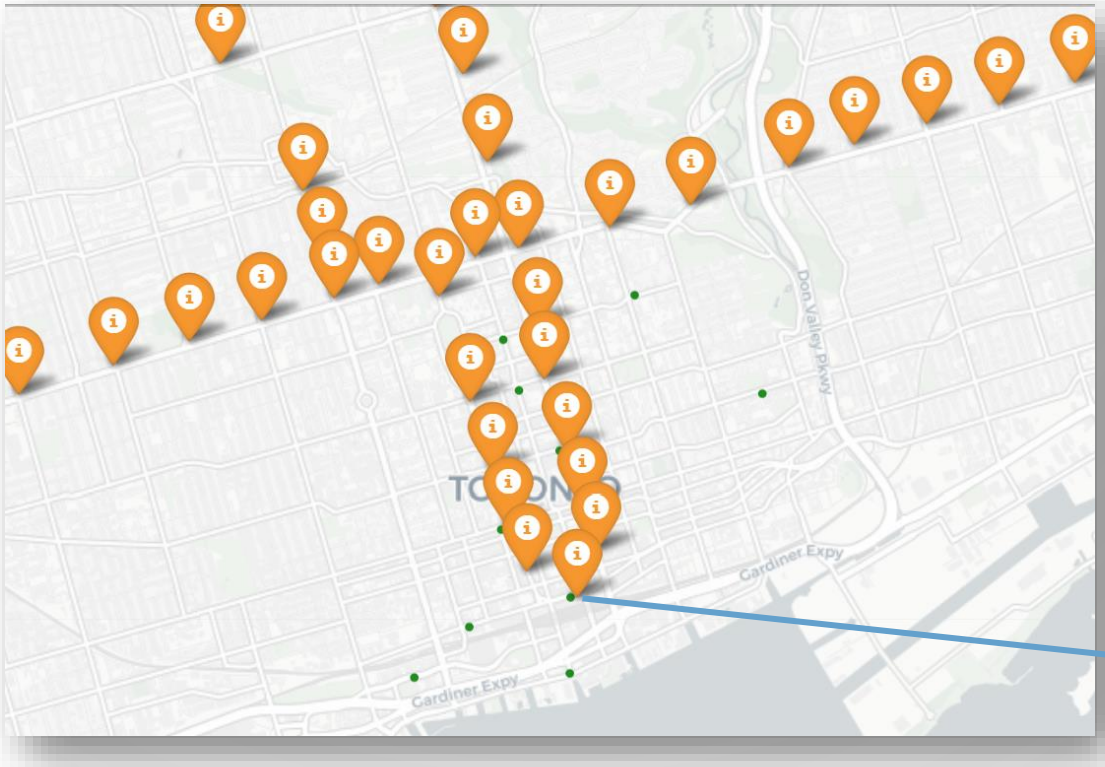
GEOSPATIAL EDA :STATIONS

Let's visualize the top 10 stations on the map !



GEOSPATIAL EDA :SUBWAY STATIONS

Which Subway Stations are closely related to bike rides?



Line 1 – Union Station

GEOSPATIAL EDA :CONCLUSION

Where?

Casual/Daily

- Downtown Toronto
  - Front St
  - Bathurst
  - Union station
- Lakeshore
  - Queens Quay
  - Lake Shore BLVD
  - HTO Park
  - Lower Simcoe

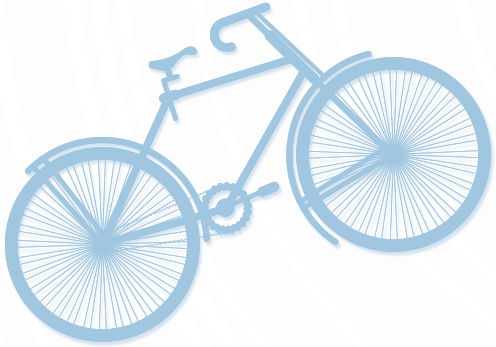
Monthly/Yearly

- Scarborough
- Etobicoke
- North York
- Downtown Toronto
  - Bay
  - University Avenue
  - Yonge
  - College
  - Wellesley
  - Dundas
  - Osgoode Station
  - Queen Station
  - College station
  - Wellesley Station

EDA :CONCLUSION

What Features to use in the Model ?

- User Type
- Tempe-rature
- Wind Chill
- Visibility
- Week of Day
- Season
- Routes
- Neigh-borhood



MODEL

INITIAL FEATURES

- We will be using the bikeshare data from 2018 to create a model
- The initial features have a weak correlation to the target variable(Trip Duration).
- We will have to engineer features that have a stronger correlation to the target variable to create a good model.

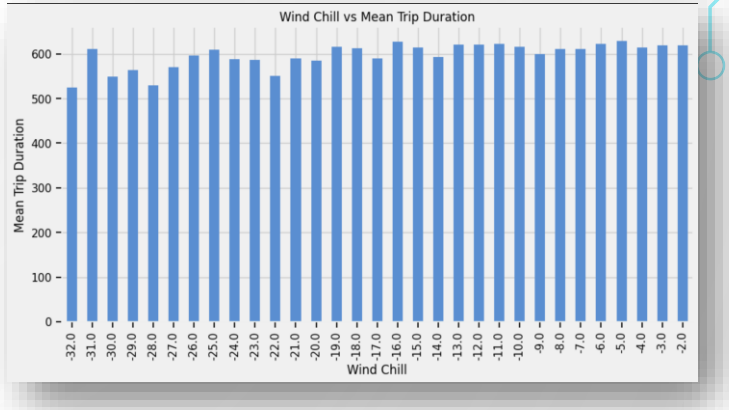
```
#View 2018 correlation for trip duration
df_merged_2018.corr()['Trip Duration'].sort_values(ascending=False)
```

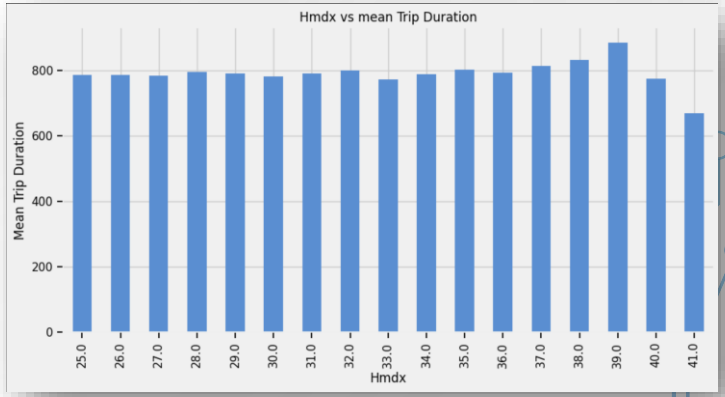
✓ 0.5s

```
Trip Duration          1.000000
Temp (°C)              0.147842
Dew Point Temp (°C)    0.131424
End Station Id         0.093238
Start Station Id       0.085047
Visibility (km)        0.029339
Wind Chill             0.015366
Hmdx                   0.004105
Rel Hum (%)           -0.009707
Trip Id               -0.012748
Wind Dir (10s deg)    -0.030016
Precip. Amount (mm)   -0.052115
Wind Spd (km/h)       -0.056237
Name: Trip Duration, dtype: float64
```

DATA ANALYSIS FOR FEATURE ENGINEERING

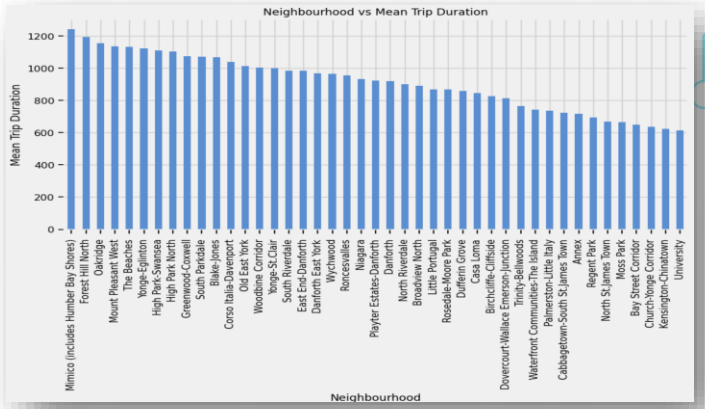
- Due to the original features not having a strong correlation with trip duration. We need to analyze the data to Engineer features that are better correlated
- The Graphs to the right show the mean trip for each unique value of windchill and humidex
- Although there isn't great variability, we can still categorically encode these variables.

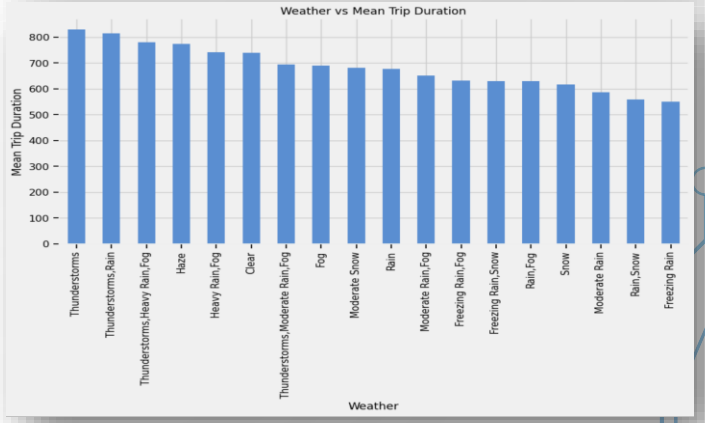




DATA ANALYSIS FOR FEATURE ENGINEERING

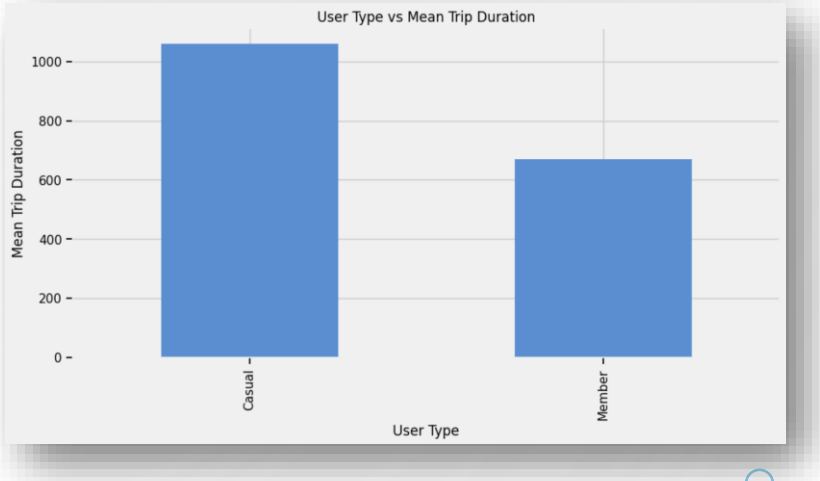
- The mean trip duration for Neighborhoods and Weather show much greater variability.
- Categorically Encoding these variables may help the model.
- Additionally, one feature to add that would logically be correlated to trip duration is distance.





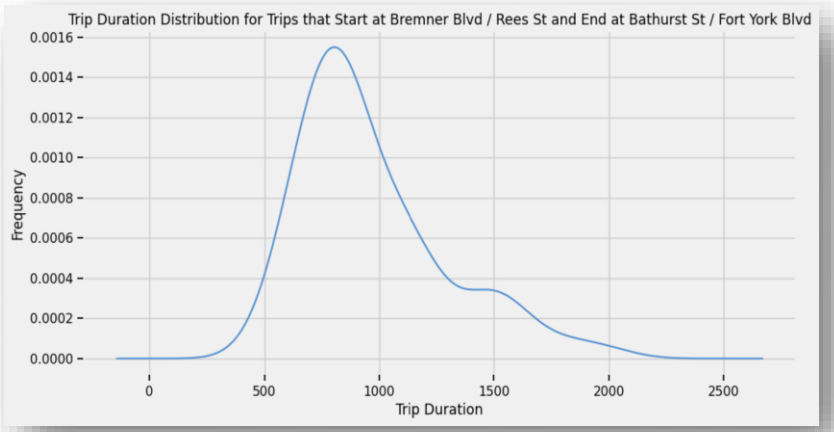
DATA ANALYSIS FOR FEATURE ENGINEERING

- There is a significant difference in mean trip duration between different types of users
- Thus, it would also be helpful to numerically encode the user type
- One thing that would also be helpful would be to categorize the different combinations of start and end stations



DATA ANALYSIS FOR FEATURE ENGINEERING

- Looking at the distribution for one possible combination of start and end stations, we can see that a large portion of trips for this combination take between 750-1000 seconds
- Therefore, it may be helpful to categorize the different trip combinations
- We will group trip combinations by increments of 100 seconds with respect to their mean trip duration



FEATURE ENGINEERING

- A pipeline will be created to add the categorical features to the dataframe.
- Due to the latitude and longitude of the start and end stations, we can also define a haversine function to calculate the distance between stations as a feature.

```
#Lets create a pipeline function that does everything we did above
#This includes: adding Hmdx Category, Wind Chill Category, Weather Category, Distance(Using Haversine formula), Combo, and Start Station Neighborhood
def pipeline(df):
    #Add Weekday column
    df['Weekday'] = df['Start Time'].dt.day_name()
    #Add Hour column
    df['Start Hour'] = df['Start Hour'].dt.hour
    #Add Hmdx Category
    df['Hmdx Category'] = df['Hmdx'].apply(lambda x: 'Extreme' if x >= 40 else ('High' if x >= 30
    else ('Moderate' if x >= 20 else ('Low' if x >= 10 else 'Very Low'))))
    #Make Hmdx Category a category type
    df['Hmdx Category'] = df['Hmdx Category'].astype('category').cat.codes
    #Add Wind Chill Category
    df['Wind Chill Category'] = df['Wind Chill'].apply(lambda x: 'Extreme' if x <= -20 else ('High' if x <= -10 else
    ('Moderate' if x <= 0 else ('Low' if x <= 10 else 'Very Low')))).astype('category').cat.codes
    #Add Weather Category
    df['Weather Category'] = df['Weather'].astype('category').cat.codes
    #Add Distance
    df['Distance'] = df.apply(lambda x: haversine(x['Start lat'], x['Start lon'], x['End lat'], x['End lon']), axis=1)
    #Add Combo
    #For this, we need to find the mean trip duration for every combination of start station, end station
    #Then we need to categorize them numerically in increments of 100 seconds
    # (Any Combo with mean trip duration between 0-100 would be 0, 100-200 would be 1, 200-300 would be 2, etc.)
    #The max possible mean trip duration is 2500 seconds, so we need 26 categories
    df['Combo'] = df.groupby(['Start Station Name','End Station Name'])['Trip Duration'].transform(lambda x: 0 if x.mean() <= 100
    else 1 if x.mean() <= 200 else 2 if x.mean() <= 300 else 3 if x.mean() <= 400 else 4 if x.mean() <= 500 else 5 if x.mean() <= 600
    else 6 if x.mean() <= 700 else 7 if x.mean() <= 800 else 8 if x.mean() <= 900 else 9 if x.mean() <= 1000 else 10 if x.mean() <= 1100 else 11
    if x.mean() <= 1200 else 12 if x.mean() <= 1300 else 13 if x.mean() <= 1400 else 14 if x.mean() <= 1500 else 15 if x.mean() <= 1600 else 16
    if x.mean() <= 1700 else 17 if x.mean() <= 1800 else 18 if x.mean() <= 1900 else 19 if x.mean() <= 2000 else 20 if x.mean() <= 2100 else 21
    if x.mean() <= 2200 else 22 if x.mean() <= 2300 else 23 if x.mean() <= 2400 else 24 if x.mean() <= 2500 else 25)
    #Add Start Station Neighborhood using map
    df = pd.merge(df, bikeshare_stations_gdf[['Station Id','neighbourhood']], left_on='Start Station Id', right_on='Station Id', how='left')
    df.drop(['Station Id'],axis=1,inplace=True)
    df.rename(columns={'neighbourhood': 'Start Station Neighborhood'}, inplace=True)
    #Make Start Station Neighborhood a category type
    df['Start Station Neighborhood'] = df['Start Station Neighborhood'].astype('category').cat.codes
    #Add User Type Category
    df['User Type Category'] = df['User Type'].astype('category').cat.codes
    #The merging added a lot of unnecessary columns, so we can specify which columns we want to keep
    df=df[['Trip Id', 'Start Time', 'End Time', 'Trip Duration','Start Station Id',
    'Start Station Name', 'End Station Id','End Station Name', 'User Type', 'Start Hour', 'Temp (°C)','Dew Point Temp (°C)',
    'Rel Hum (%)', 'Wind Dir (10s deg)','Wind Spd (km/h)', 'Visibility (km)', 'Hmdx', 'Wind Chill', 'Weather','Precip. Amount (mm)',
    'Weekday', 'Start lat', 'Start lon', 'End lat', 'End lon', 'Hmdx Category','Distance', 'Combo',
    'Weather Category','Start Station Neighborhood', 'User Type Category', 'Wind Chill Category']]
    return df
✓ 0.7s
```

FEATURE CORRELATION TO TRIP DURATION

- We can see that the new features have a strong correlation to our target variable.
- In particular, Trip Combination, Distance and the User Type Category have the strongest correlations. Their correlation coefficients are; 0.802, 0.444, and -0.356 respectively.

```
#View the Correlation for every feature of 2018 to trip duration
df_merged_2018_train.corr()['Trip Duration'].sort_values(ascending=False)
```

✓ 3.7s

```
Output exceeds the size limit. Open the full output data in a text editor
Trip Duration                          1.000000
Combo                                  0.801911
Distance                               0.444349
End Station Id                         0.149261
Start Station Id                       0.138606
Temp (°C)                              0.130598
Dew Point Temp (°C)                    0.112891
Start Station Neighborhood Category    0.112321
Rush Hour                              0.094941
Trip Id                                0.086692
Wind Chill Category                    0.078667
Start Hour                             0.068007
capacity_end                           0.063151
capacity_start                         0.058427
Visibility (km)                        0.033331
Wind Chill                             0.012326
Hmdx                                   0.003068
Rel Hum (%)                           -0.005430
Wind Dir (10s deg)                    -0.022786
Weather Category                      -0.033483
Wind Spd (km/h)                       -0.054220
Precip. Amount (mm)                   -0.073104
Hmdx Category                         -0.077941
lon_end                               -0.092254
lon_start                             -0.092933
...
Weekday                               -0.125798
lat_start                             -0.128660
lat_end                               -0.133003
User Type Cat                         -0.353608
Name: Trip Duration, dtype: float64
```

MACHINE LEARNING MODELS

- We will try three types of model to predict trip duration.
- Random Forest, Linear Regression, and Light Gradient Boosting.
- To optimize the hyperparameters, we will use BayesSearchCV. As it is not as computationally extensive as GridSearchCV.
- We will train the data on the first 6 months of 2018, and test on the last 6 months.

LIGHT GRADIENT BOOSTING MODEL

- The Light Gradient Boosting Model has its hyperparameters optimized with BayesSearchCV.

```
#Lets Use Bayesian Optimization to find the best parameters for LightGBM
#Bayesian Optimization is a method of hyperparameter optimization that uses a probabilistic model to optimize the hyperparameters of a machine learning model
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.utils import use_named_args
from sklearn.model_selection import cross_val_score
#Define the search space
#Cannot use bagging in GOSS
search_spaces = {'learning_rate': Real(0.01, 1.0, 'log-uniform'),
                 'num_leaves': Integer(2, 100),
                 'max_depth': Integer(1, 100),
                 'min_child_samples': Integer(1, 100),
                 'max_bin': Integer(100, 1000),
                 'subsample': Real(0.01, 1.0, 'uniform'),
                 'subsample_freq': Integer(0, 10),
                 'colsample_bytree': Real(0.01, 1.0, 'uniform'),
                 'min_child_weight': Real(0, 10),
                 'subsample_for_bin': Integer(100000, 500000),
                 'reg_alpha': Real(0, 1.0),
                 'reg_lambda': Real(0, 1.0),
                 'n_estimators': Integer(100, 1000),
                 'boosting_type': Categorical(['gbdt', 'dart']),
                 'objective': Categorical(['regression']),
                 'random_state': Integer(1, 1000),
                 'n_jobs': Integer(1, 10),
                 'silent': Categorical([True]),
                 'importance_type': Categorical(['split', 'gain'])}
#Define the model
lgb_model = lgb.LGBMRegressor()
#Define the search
search = BayesSearchCV(estimator=lgb_model, search_spaces=search_spaces, n_iter=32, cv=5, n_jobs=-1, verbose=0, refit=True, random_state=1)
#Fit the search
result = search.fit(X_train, y_train)
#Print the best parameters
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)
#Print the evaluation metrics for test
y_pred = result.predict(X_test)
print('R^2:',r2_score(y_test,y_pred))
print('RMSE:',np.sqrt(mean_squared_error(y_test,y_pred)))
print('MAE:',mean_absolute_error(y_test,y_pred))
```

LINEAR REGRESSION MODEL

- The Linear Regression model and its hyperparameter optimization.

```
#Lets also run a BayesSearchCV on Linear Regression
from sklearn.linear_model import LinearRegression
#Define the search space
search_spaces = {'fit_intercept': Categorical([True, False]),
                 'normalize': Categorical([True, False]),
                 'copy_X': Categorical([True, False]),
                 'n_jobs': Integer(1, 10)}
#Define the model
lr_model = LinearRegression()
#Define the search
search = BayesSearchCV(estimator=lr_model, search_spaces=search_spaces, n_iter=32, cv=5, n_jobs=-1, verbose=0, refit=True, random_state=42)
#Fit the search
#Use the same train and test as Random Forest since that does not have NaN values
result = search.fit(X_train_rf, y_train_rf)
#Print the best parameters
print('Best parameters:', result.best_params_)
#Print the best score
print('Best score:', result.best_score_)
#Print the best model
lr_model = LinearRegression(**result.best_params_)
lr_model.fit(X_train_rf,y_train_rf)
y_pred = lr_model.predict(X_test_rf)
print('R^2:',r2_score(y_test,y_pred))
print('RMSE:',np.sqrt(mean_squared_error(y_test,y_pred)))
print('MAE:',mean_absolute_error(y_test,y_pred))
```

✓ 2m 44.3s

RANDOM FOREST MODEL

- The Random Forest model and its hyperparameter optimization.

```python
#Use BayesSearchCV to find the best parameters for the Random Forest Regressor
import skopt as skopt
from skopt import BayesSearchCV
#Define the parameter space
param_space = {
    'n_estimators': (10, 1000),
    'max_depth': (1, 100),
    'min_samples_split': (2, 10),
    'min_samples_leaf': (1, 10),
    'max_features': (0.1, 1.0),
    'max_samples': (0.1, 1.0)
}
#Create the model
rf_model = RandomForestRegressor()
#Create the BayesSearchCV object
bayes_cv_tuner = BayesSearchCV(
    estimator = rf_model,
    search_spaces = param_space,
    n_iter = 10,
    cv = 3,
    n_jobs = -1,
    verbose = 0,
    refit = True,
    random_state = 42
)
#Fit the model
bayes_cv_tuner.fit(X_train_rf, y_train_rf)
#Print the best parameters
print(bayes_cv_tuner.best_params_)
#Print the best score
print(bayes_cv_tuner.best_score_)
#Print the best estimator
print(bayes_cv_tuner.best_estimator_)
#Predict on the test set
y_pred = bayes_cv_tuner.predict(X_test_rf)
#Print the evaluation metrics
print('R^2:',r2_score(y_test,y_pred))
print('RMSE:',np.sqrt(mean_squared_error(y_test,y_pred)))
print('MAE:',mean_absolute_error(y_test,y_pred))
#Print the evaluation metrics for train\
y_pred_train = bayes_cv_tuner.predict(X_train_rf)
print('R^2:',r2_score(y_train_rf,y_pred_train))
print('RMSE:',np.sqrt(mean_squared_error(y_train_rf,y_pred_train)))
print('MAE:',mean_absolute_error(y_train_rf,y_pred_train))
```

✓ 279m 28.9s

RESULTS OF THE MODELS

- The metrics that will be used to measure the success of each model will be the R-Squared Score, The Root Mean Square Error, and the Mean Absolute Error
- The Light Gradient Boosting model has the best scores for every metric.
- Thus, we will use the Light Gradient Boosting Model to predict the trip duration for future years.

	Model	R^2	RMSE	MAE
0	LightGBM	0.753652	182.693621	116.618659
1	Linear Regression	0.741736	191.596761	126.557863
2	Random Forest	0.741834	191.560359	129.436315

LIGHT GRADIENT BOOSTING EXPLAINED

- LGB is a gradient boosting framework based on decision trees that has an increased efficiency which also reduces memory usage.
- The Efficiency is increased due to Exclusive Feature Bundling, which identifies mutually exclusive features and merges them.
- LGB also uses gradient based one side sampling, which essentially resamples instances with small error, but retains ones with moderate-high training error.

LIMITATIONS OF THE DATASET

- The initial features are not great predictors of the target variable. Making feature engineering more difficult.
- If we were provided features such as; age, trip purpose, user ID, the model would be a lot more accurate.