Google Data Analytics Course Capstone Project: Case Study 1 "Cyclistic"

This article is my approach and work to solve the problem of Google Data Analytics Course Capstone Project: Case Study 1 "Cyclistic". The main objective of our case study is "**How to convert casuals to members?"** or to be specific, a successful bike-sharing company desires to increase the number of their annual memberships. You can find the full details of the case study here.

As I learned from the Google Data Analytics program, I will follow the steps of the data analysis process: **ask**, **prepare**, **process**, **analyze, share and act.** However, since act step is for executives to decide, I will not cover that step here.

Ask

The questions that needs to be answered are:

1. How do annual members and casual riders use Cyclistic bikes differently?
2. Why would casual riders buy Cyclistic annual memberships?
3. How can Cyclistic use digital media to influence casual riders to become members?

Prepare

In this step, we prepare the data by obtaining the dataset and storing it. The datasets are given as a monthly based trip data in a .zip file. I downloaded the last 12 months of trip data as 12 different .zip files and extracted them. We don't need to mine or scrape the data, its given as a .csv file for each month.

Process

In this step we process the data and prepare it for our next step where we will find answers to our questions. I used PySpark SQL with Jupyter Notebook for this step since the dataset is too large to merge and operate (around 4,073,561 observations). PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing your data in a distributed environment. PySpark supports most of Spark's features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) and Spark Core.[1]

At first, I tried to merge and process the 12 .csv files with R-studio on my pc (not cloud), however it took too much time and crashed after some point. Let's import necessary packages such as Datatypes and functions, then create a spark session:

```
from pyspark.sql import SparkSession
```

```
from pyspark.sql.types import *
```

```
from pyspark.sql.functions import *
```

```
spark=SparkSession.builder.getOrCreate()
```

Next, let's read and merge all .csv files into one large dataset:

```
ds1=spark.read.csv('202105-divvy-tripdata.csv', header=True)
ds2=spark.read.csv('202104-divvy-tripdata.csv', header=True)
ds3=spark.read.csv('202103-divvy-tripdata.csv', header=True)
ds4=spark.read.csv('202102-divvy-tripdata.csv', header=True)
ds5=spark.read.csv('202101-divvy-tripdata.csv', header=True)
ds6=spark.read.csv('202012-divvy-tripdata.csv', header=True)
ds7=spark.read.csv('202011-divvy-tripdata.csv', header=True)
ds8=spark.read.csv('202010-divvy-tripdata.csv', header=True)
ds9=spark.read.csv('202009-divvy-tripdata.csv', header=True)
ds10=spark.read.csv('202008-divvy-tripdata.csv', header=True)
ds11=spark.read.csv('202007-divvy-tripdata.csv', header=True)
ds12=spark.read.csv('202006-divvy-tripdata.csv', header=True)
```

Let's observe the number of rows and columns:

```
print((ds.count(), len(ds.columns)))
```

```
(4073561, 13)
```

As you see our dataset become very large with more than 4 million rows and 13 columns. Now let's have a peek to the dataset using first(), which shows only the first row and show():

```
ds.first()
```

```
Row(ride_id='C809ED75D6160B2A', rideable_type='electric_bike', started_at='2021-05-30 11:58:15', ended_at='2021-05-30
12:10:39', start_station_name=None, start_station_id=None, end_station_name=None, end_station_id=None, start_lat='41.
9', start_lng='-87.63', end_lat='41.89', end_lng='-87.61', member_casual='casual')
```

```
ds.show(5)
```

```
+----------------+-------------+-------------------+-------------------+------------------+----------------+---------
-------+--------------+---------+---------+-------+-------+-------------+
|         ride_id|rideable_type|         started_at|           ended_at|start_station_name|start_station_id|end_stati
on_name|end_station_id|start_lat|start_lng|end_lat|end_lng|member_casual|
+----------------+-------------+-------------------+-------------------+------------------+----------------+---------
-------+--------------+---------+---------+-------+-------+-------------+
|C809ED75D6160B2A|electric_bike|2021-05-30 11:58:15|2021-05-30 12:10:39|              null|            null|
null|          null|     41.9|   -87.63|  41.89| -87.61|       casual|
|DD59FDCE0ACACAF3|electric_bike|2021-05-30 11:29:14|2021-05-30 12:14:09|              null|            null|
null|          null|    41.88|   -87.62|  41.79| -87.58|       casual|
|0AB83CB88C43EFC2|electric_bike|2021-05-30 14:24:01|2021-05-30 14:25:13|              null|            null|
null|          null|    41.92|    -87.7|  41.92|  -87.7|       casual|
|7881AC6D39110C60|electric_bike|2021-05-30 14:25:51|2021-05-30 14:41:04|              null|            null|
null|          null|    41.92|    -87.7|  41.94| -87.69|       casual|
|853FA701B4582BAF|electric_bike|2021-05-30 18:15:39|2021-05-30 18:22:32|              null|            null|
null|          null|    41.94|   -87.69|  41.94|  -87.7|       casual|
+----------------+-------------+-------------------+-------------------+------------------+----------------+---------
-------+--------------+---------+---------+-------+-------+-------------+
only showing top 5 rows
```

We have 13 columns and we can infer their content:

- **ride_id**: Id for each trip taken, as of now we are not sure if they are unique or not, we have to find out
- **rideable_type**: Represents the type of a bike
- **started_at**: Date and time of the start time
- **ended_at**: Date and time of the end time
- **start_station_name**: Name of the starting station
- **start_station_id**: Id of the starting station
- **end_station_name**: Name of the ending station
- **end_station_id**: Id of the ending station
- **start_lat**: Latitude of the starting point
- **start_lng**: Longitude of the starting point
- **end_lat**: Latitude of the ending point
- **end_lng**: Longitude of the ending point
- **member_casual**: Represents the membership status

Let's remove duplicate rows using dropDuplicates() function and then count all rows again:

```
ds.dropDuplicates().count()
```

```
4073561
```

Since the number of rows did not change, it means that ride_id is unique for each row. Next let's create a column with a name **distance_traveled** where I will calculate the distance for each trip in **meters** using start and end lat-lng. For this, we can use Haversine formula, where it calculates the distance between two points given latitude and longitude of both points. [2] I found the ported pyspark version of the Haversine formula from the github. [3] Also, there are online Haversine formula calculators, such as this.

```
import pyspark.sql.functions as F
ds = ds.withColumn("a", (
        F.pow(F.sin(F.radians(F.col("end_lat") - F.col("start_lat")) / 2), 2) +
        F.cos(F.radians(F.col("start_lat"))) * F.cos(F.radians(F.col("end_lat"))) *
        F.pow(F.sin(F.radians(F.col("end_lng") - F.col("start_lng")) / 2), 2)
    )).withColumn("distance_traveled", F.atan2(F.sqrt(F.col("a")), F.sqrt(-F.col("a") + 1)) * 12742000)
```

Let's have a peek at the distance_traveled column:

```
ds.select('start_lat','start_lng', 'end_lat', 'end_lng','distance_traveled').show(5)
```

```
+---------+---------+-------+-------+------------------+
|start_lat|start_lng|end_lat|end_lng| distance_traveled|
+---------+---------+-------+-------+------------------+
|     41.9|   -87.63|  41.89| -87.61|1994.1890794035653|
|    41.88|   -87.62|  41.79| -87.58|10541.961176829122|
|    41.92|    -87.7|  41.92|  -87.7|               0.0|
|    41.92|    -87.7|  41.94| -87.69| 2372.775481761235|
|    41.94|   -87.69|  41.94|  -87.7| 827.1180459695228|
+---------+---------+-------+-------+------------------+
only showing top 5 rows
```

Next, I would like to find out the time each trip took. For this, I created a new column named **date_diff** where using the started_at and ended_at columns, I can find the day difference. To do this, I used .datediff and .to_date functions of pyspark:

```
#create a column that finds the date difference, and show it in descending order (largest to least)
ds= ds.withColumn('date_diff', F.datediff(F.to_date(ds.ended_at), F.to_date(ds.started_at)))
ds.select('started_at', 'ended_at', 'member_casual','date_diff').sort(ds.date_diff.desc()).show(10)
```

```
+-------------------+-------------------+-------------+---------+
|         started_at|           ended_at|member_casual|date_diff|
+-------------------+-------------------+-------------+---------+
|2020-09-02 18:34:33|2020-10-10 11:17:54|       casual|       38|
|2021-05-02 02:56:07|2021-06-08 13:37:43|       casual|       37|
|2020-09-06 23:20:29|2020-10-12 11:46:25|       casual|       36|
|2020-09-05 08:50:15|2020-10-10 13:43:02|       casual|       35|
|2020-07-05 14:25:39|2020-08-09 07:11:06|       casual|       35|
|2020-07-05 01:51:06|2020-08-08 12:13:19|       casual|       34|
|2020-07-02 17:26:55|2020-08-04 07:16:12|       casual|       33|
|2021-04-02 17:50:00|2021-05-05 22:06:42|       casual|       33|
|2020-07-02 19:49:10|2020-08-04 18:00:37|       casual|       33|
|2020-07-07 14:36:11|2020-08-09 19:13:11|       casual|       33|
+-------------------+-------------------+-------------+---------+
only showing top 10 rows
```

We can see from the descending order of date difference that the maximum amount of time spent for a trip is 38 days, during 2 Sep 2020 - 10 Oct 2020. It is interesting that all of these outliers are not members but only casuals. Now, I sorted same columns in an ascending order to observe that we have negative date difference, which is impossible:

```
#when we order the date difference in ascending order, we see that we have incorrect negative values
ds.select('started_at', 'ended_at', 'member_casual','date_diff').sort(ds.date_diff.asc()).show(10)
```

```
+-------------------+-------------------+-------------+---------+
|         started_at|           ended_at|member_casual|date_diff|
+-------------------+-------------------+-------------+---------+
|2020-12-15 11:35:17|2020-11-25 09:15:01|       member|      -20|
|2020-12-15 11:52:50|2020-11-25 15:31:09|       member|      -20|
|2020-12-15 11:55:15|2020-11-25 16:14:08|       member|      -20|
|2020-12-15 11:41:33|2020-11-25 11:46:44|       member|      -20|
|2020-12-15 11:42:13|2020-11-25 12:01:32|       member|      -20|
|2020-12-15 11:58:22|2020-11-25 16:40:43|       member|      -20|
|2020-12-15 12:09:07|2020-11-25 18:58:46|       casual|      -20|
|2020-12-15 12:12:08|2020-11-25 21:46:45|       member|      -20|
|2020-12-15 11:55:57|2020-11-25 16:13:35|       member|      -20|
|2020-12-15 11:45:32|2020-11-25 13:24:26|       member|      -20|
+-------------------+-------------------+-------------+---------+
only showing top 10 rows
```

If we observe closely, a member took a trip on 15 Dec 2020 to 25 Nov 2020. Having a bachelor degree from Physics, I learned that time machine was not invented yet and it's impossible to time travel in theory. All jokes aside, these rows clearly indicate wrong input and should be removed from the dataset:

```
ds=ds.filter(col('date_diff').cast(LongType()) >= 0)
ds.count()
```

```
4073182
```

Once removed, we now have 4,073,182 rows which is 379 rows less than the beginning. We can run the same code to observe the min day difference is 0 with only time difference. Now, once we have the day differences, we calculate time differences to find the duration **in minutes** for each trip using .minute, .second and .hour functions of pyspark:

```
ds = ds.withColumn('duration_in_min', (ds.date_diff*24*60) + F.hour(ds.ended_at)*60+F.minute(ds.ended_at)+
      F.second(ds.ended_at)/60- F.hour(ds.started_at)*60-F.minute(ds.started_at)- F.second(ds.started_at)/60)
```

Let's have a peek at some columns from the dataset, including the new column **duration_in_min**:

```
ds.select('started_at', 'ended_at', 'member_casual','date_diff', 'distance_traveled', 'duration_in_min').show(10)
```

```
+-------------------+-------------------+-------------+---------+------------------+------------------+
|         started_at|           ended_at|member_casual|date_diff| distance_traveled|   duration_in_min|
+-------------------+-------------------+-------------+---------+------------------+------------------+
|2021-05-30 11:58:15|2021-05-30 12:10:39|       casual|        0|1994.1890794035653|12.399999999999977|
|2021-05-30 11:29:14|2021-05-30 12:14:09|       casual|        0|10541.961176829122| 44.91666666666664|
|2021-05-30 14:24:01|2021-05-30 14:25:13|       casual|        0|               0.0|1.2000000000000304|
|2021-05-30 14:25:51|2021-05-30 14:41:04|       casual|        0| 2372.775481761235| 15.21666666666672|
|2021-05-30 18:15:39|2021-05-30 18:22:32|       casual|        0| 827.1180459695228| 6.883333333333303|
|2021-05-30 11:33:41|2021-05-30 11:57:17|       casual|        0|1386.2667237514747| 23.59999999999997|
|2021-05-30 10:51:37|2021-05-30 11:06:20|       casual|        0|3436.9377567631027|14.716666666666704|
|2021-05-05 13:57:03|2021-05-05 14:14:58|       casual|        0|               0.0|17.916666666666696|
|2021-05-05 11:31:26|2021-05-05 11:34:03|       casual|        0|               0.0| 2.616666666666621|
|2021-05-04 19:51:05|2021-05-04 20:17:26|       casual|        0|               0.0|26.350000000000062|
+-------------------+-------------------+-------------+---------+------------------+------------------+
only showing top 10 rows
```

We can check if the duration of each trip in minutes is correct using started_at and ended_at columns, which it is. So far, we have calculated distance (in meters) and duration (in minutes) for each trip. Let's see the summary of those values:

```
ds.select('distance_traveled','duration_in_min').summary().show()
```

```
+-------+------------------+------------------+
|summary| distance_traveled|   duration_in_min|
+-------+------------------+------------------+
|  count|           4068146|           4073182|
|   mean|   2221.8403058682| 26.88242978094281 5|
| stddev|2025.9532818899693|  236.6782296468922|
|    min|               0.0|-120.30000000000004|
|    25%| 865.4421767937478|  7.66666666666621|
|    50%|1675.2243117725307| 14.016666666666605|
|    75%|3018.4568996381654| 25.850000000000016|
|    max| 48370.80097108494|          54283.35|
+-------+------------------+------------------+
```

We can spot a problem from above summary that the duration of a trip can't be a negative value. Thus, we have to check it further by sorting duration in an ascending order:

```
ds.select('started_at', 'ended_at', 'date_diff', 'duration_in_min').sort(ds.duration_in_min.asc()).show(10)
```

```
+-------------------+-------------------+---------+-------------------+
|         started_at|           ended_at|date_diff|    duration_in_min|
+-------------------+-------------------+---------+-------------------+
|2020-07-25 15:08:21|2020-07-25 13:08:03|        0|-120.30000000000004|
|2020-10-16 16:44:52|2020-10-16 15:09:51|        0| -95.01666666666664|
|2020-10-16 16:44:53|2020-10-16 15:10:54|        0| -93.98333333333336|
|2020-10-16 16:44:55|2020-10-16 15:11:27|        0| -93.46666666666663|
|2020-10-16 16:44:56|2020-10-16 15:43:14|        0| -61.69999999999998|
|2020-10-16 16:44:58|2020-10-16 15:45:03|        0|-59.916666666666714|
|2020-11-01 01:57:30|2020-11-01 01:00:39|        0|             -56.85|
|2020-11-01 01:56:26|2020-11-01 01:00:29|        0|-55.949999999999996|
|2020-11-01 01:55:57|2020-11-01 01:02:04|        0| -53.88333333333333|
|2020-11-01 01:54:40|2020-11-01 01:01:34|        0|-53.099999999999994|
+-------------------+-------------------+---------+-------------------+
only showing top 10 rows
```

According to above results, somebody rented a bike starting at 15:08:21 to 13:08:03 on 25 July 2020 giving us a negative duration. Again, these are the observations we have to remove:

```
ds=ds.filter(col('duration_in_min') >= 0.0)
ds.count()
```

```
4063030
```

According to above code, around 2k rows were removed from the dataset. Now, let's create a column **day_of_week** which will represent the day of the trip. To do this, we can use date_format function of the pyspark:

t

```
ds=ds.withColumn("day_of_week", date_format(col('started_at'), 'EEEE'))
```

Now let's observe the distribution for some categorical (string) columns:

```python
#casual vs member distribution
ds.cube('member_casual').count().show()
```

```
+-------------+-------+
|member_casual|  count|
+-------------+-------+
|       casual|1710107|
|         null|4063030|
|       member|2352923|
+-------------+-------+
```

```python
#frequency distribution of day of the week
ds.cube('day_of_week').count().show()
```

```
+-----------+-------+
|day_of_week|  count|
+-----------+-------+
|  Wednesday| 529720|
|   Thursday| 533708|
|    Tuesday| 503792|
|     Monday| 503884|
|       null|4063030|
|     Friday| 597210|
|     Sunday| 637741|
|   Saturday| 756975|
+-----------+-------+
```

```python
#frequency distr of bike types
ds.cube('rideable_type').count().show()
```

```
+-------------+-------+
|rideable_type|  count|
+-------------+-------+
| classic_bike| 843555|
|         null|4063030|
|electric_bike| 888224|
|  docked_bike|2331251|
+-------------+-------+
```

I used .cube function to show the frequency distributions. Also, **null** represents the total number of rows in the dataset.

We can observe that there are more members than casuals. Also, busiest day of week is Saturday. Furthermore, there 3 types of bikes available such as classic bike, electric bike and docked bike. Where docked bike is the most popular one among three.

We will do further detailed analysis in the next step, but there is one more thing I observed.

Dataset has considerable amount of NA values especially in **start_station_name** and **end_station_name** columns.

Let's count the number of NA rows only for start_station_name column:

```
#start_station_name frequency is counted and sorted in a descending way
df=spark.createDataFrame(ds.cube('start_station_name').count().collect())
```

```
df.sort(df['count'].desc()).show(10)
```

```
+--------------------+-------+
|  start_station_name|  count|
+--------------------+-------+
|                null|4063030|
|                null| 201925|
|Streeter Dr & Gra...|  48000|
|Lake Shore Dr & M...|  39335|
|   Clark St & Elm St|  35982|
| Theater on the Lake|  35935|
|Lake Shore Dr & N...|  32401|
|     Millennium Park|  30385|
|Wells St & Concor...|  29948|
|Michigan Ave & Oa...|  28524|
+--------------------+-------+
only showing top 10 rows
```

As we can see above, there are 201,925 NA values in start_station_name column which is around 5% of total dataset. Instead of removing them, I assigned to all NA values as **missing_data** so that we can analyze missing values as well.

```
#filling all null values with missing_data value, so that we can infer smth in EDA
ds=ds.na.fill('missing_data')
```

Next, I sorted the whole dataset according to the date:

```
ds=ds.orderBy('started_at')
```

```
print((ds.count(), len(ds.columns)))
```

```
(4063030, 18)
```

The final dataset has 4,063,030 rows and 18 columns. I exported the dataset as a csv file which had large size of almost 1gb. Instead, I removed few columns that we won't be using in the analysis step of our case study:

```
ds=ds.drop('ride_id')
ds=ds.drop('start_station_id')
ds=ds.drop('end_station_id')
ds=ds.drop('start_lat')
ds=ds.drop('end_lat')
ds=ds.drop('start_lng')
ds=ds.drop('end_lng')
ds=ds.drop('a')
ds=ds.drop('date_diff')
```

```
ds.repartition(1).write.csv("ds_dropped.csv", header=True)
```

Final csv file size was around 600mb, still big but better than previous one.

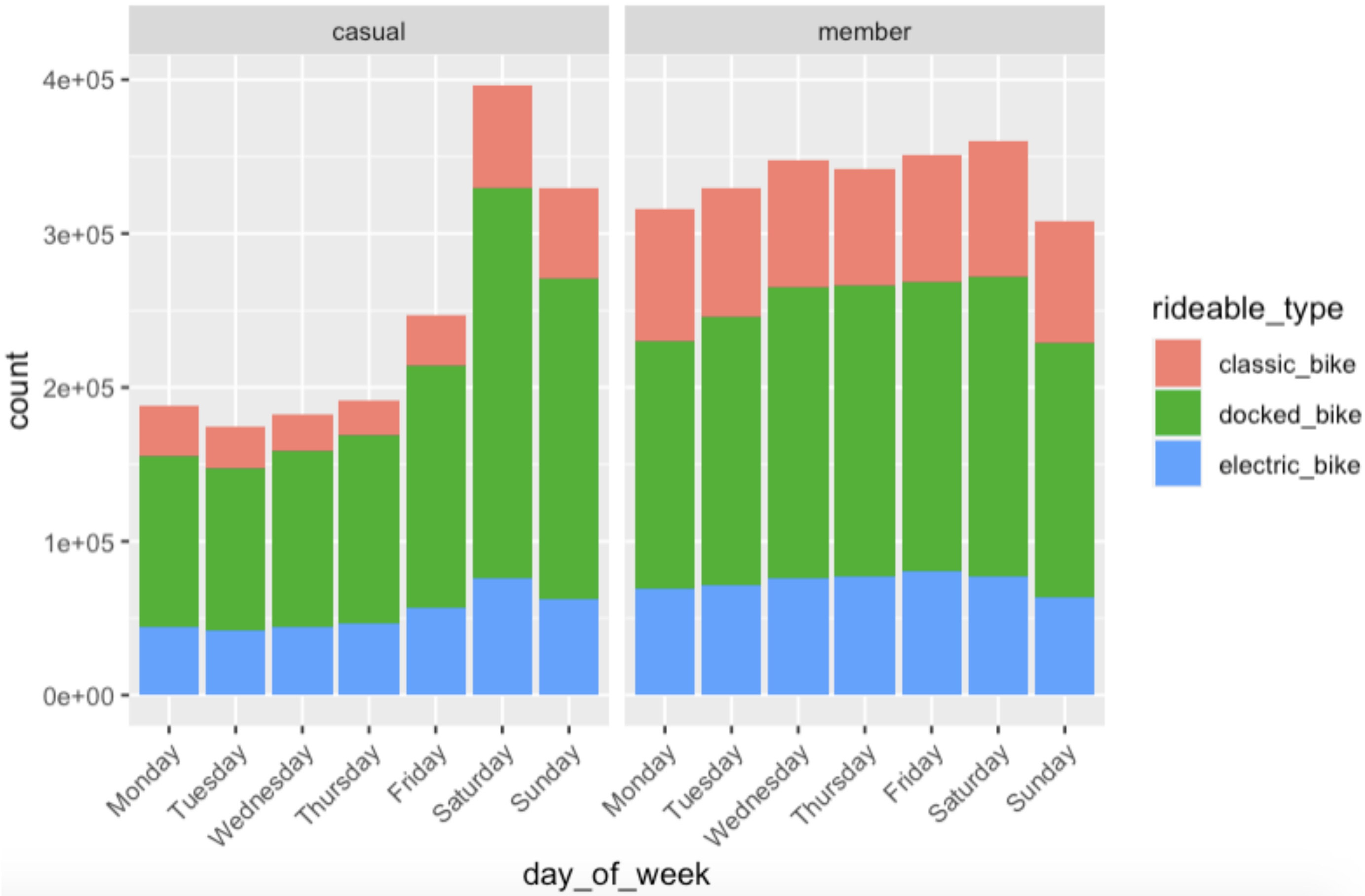
Analyze

In this step we will analyze our processed or cleaned data using R studio. If you wonder why not continue with pyspark, pyspark has no plotting, it can be done using panda library which worked slower than R. Firstly, I loaded necessary libraries and read the csv file:

```
#loading the libraries
library(tidyverse)
library(dplyr)
library(readr)
ds<-read.csv("ds_dropped.csv", header = T)
```

Now, let's plot a bar graph that shows weekly frequency distribution of the member and casual customers with bike types. For this I organized days in order from Monday to Sunday. Then applied geom_bar and fill with rideable_type.

```
#freq of day of week according to member and casual with rideable bike types
ds$day_of_week<-factor(ds$day_of_week, levels = c("Monday", "Tuesday", "Wednesday",
                                                  "Thursday","Friday","Saturday","Sunday"))
ggplot(ds)+geom_bar(mapping=aes(x=day_of_week, fill=rideable_type))+
  facet_wrap(~member_casual)+
  theme(axis.text.x=element_text(angle=45, hjust=1))
```

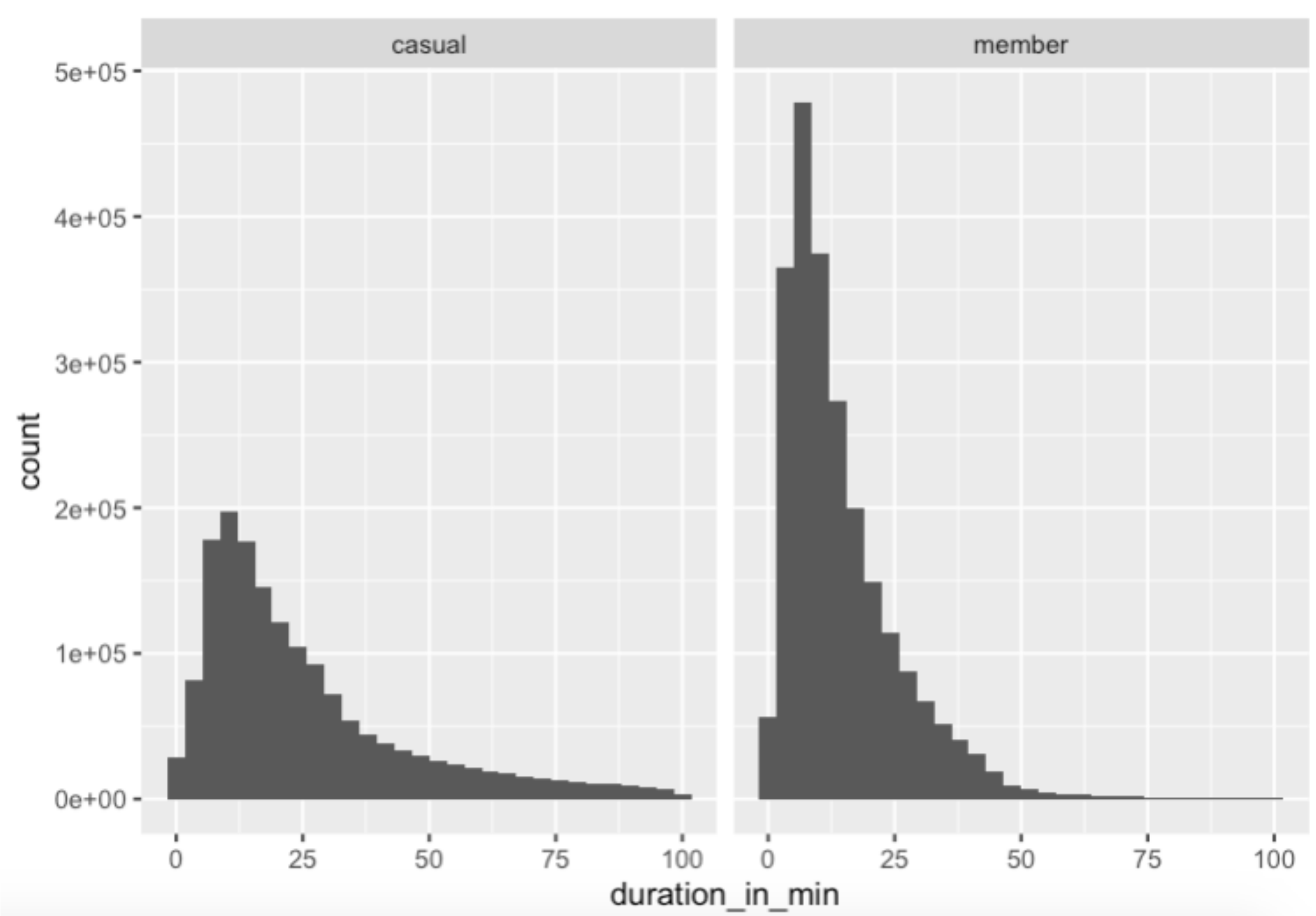


This plot shows us few observations about members and casuals. Some of them are:

- Members usage are quite similar throughout the week except Sunday. We can infer that members are mostly working people.
- Casual usage is slow for weekdays but weekends are very popular especially Saturday.
- Docked bike is the most popular for both members and casuals. But we can see that casuals prefer docked bike more than members do.

Now let's observe trip duration behavior for member and casuals. For this I used geom_histogram and filtered the duration times to less than 100 minutes for better plot:

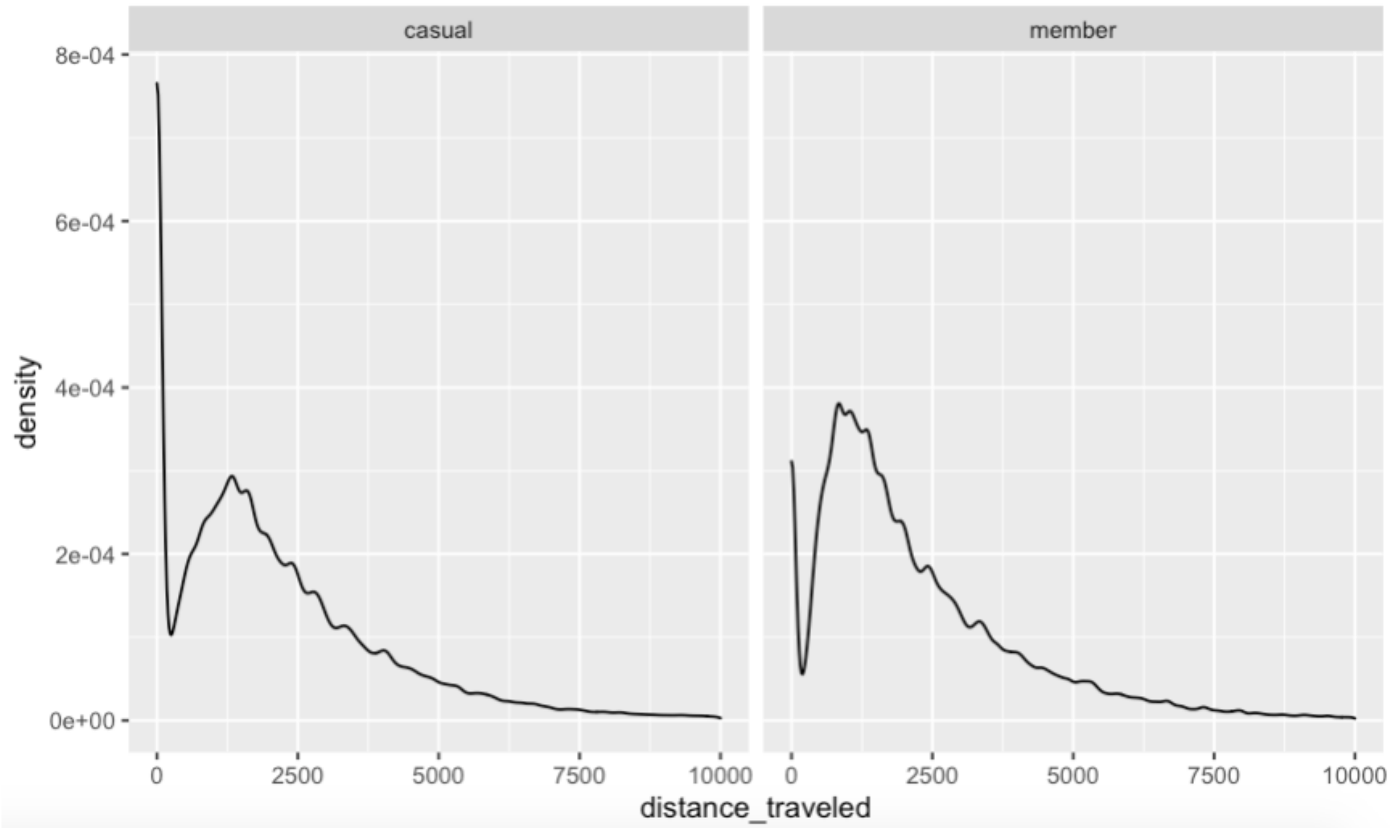
```
#duration vs distance line graph for member and casual
ggplot(filter(ds, ds$duration_in_min<100 ))+
  geom_histogram(mapping=aes(x=duration_in_min))+
  facet_wrap(~member_casual)
```



Only observation here is that members tend to take short trips than casuals. Or casuals take longer trips than members. We will talk about the mean trip duration later using summary function.

Now let's plot distance traveled in meters for casuals and members. Here I used geom_density and filtered the distance to less than 10,000 meters for better plot.

```
#distance traveled in meters
ggplot(filter(ds, ds$distance_traveled<10000 ))+
  geom_density(mapping=aes(x=distance_traveled))+
  facet_wrap(~member_casual)
```



It is hard the make any observations using this plot.

Next, I filtered dataset into two, according to member-casual status. Then applied summary function to numeric columns only to get some details. Below is the summary for members dataset.

```
#filtering only members to a new dataset
filtered_member<-filter(ds, member_casual=="member")

#summary for member ds after dropping NA values, we can observe mean, median
summary(drop_na(select(filtered_member, c('day_of_week', 'distance_traveled', 'duration_in_min'))))
```

```
    day_of_week     distance_traveled duration_in_min
 Monday   :315143   Min.   :    0.0   Min.   :    0.00
 Tuesday  :329035   1st Qu.:  940.5   1st Qu.:    6.32
 Wednesday:346757   Median : 1687.0   Median :   11.07
 Thursday :341550   Mean   : 2248.1   Mean   :   15.26
 Friday   :350229   3rd Qu.: 3018.5   3rd Qu.:   19.30
 Saturday :360218   Max.   :48370.8   Max.   :41271.00
 Sunday   :307507
```

Now, I did the same for casuals:

```
filtered_casual<-filter(ds, member_casual=="casual")
summary(drop_na(select(filtered_casual, c('day_of_week', 'distance_traveled', 'duration_in_min'))))
```

```
    day_of_week      distance_traveled duration_in_min
 Monday   :188152   Min.   :    0.0   Min.   :    0.00
 Tuesday  :174145   1st Qu.:  714.5   1st Qu.:   11.03
 Wednesday:182292   Median : 1660.8   Median :   20.20
 Thursday :191432   Mean   : 2185.4   Mean   :   41.83
 Friday   :246275   3rd Qu.: 3018.5   3rd Qu.:   38.35
 Saturday :395857   Max.   :33800.2   Max.   :54283.35
 Sunday   :329438
```

From above summary, we can observe that mean distance traveled by members and casuals are almost same, however, members mean trip duration ~15 min. is almost three times less than casual mean trip duration ~42 min.

Next, let's see the most popular start and end stations with their frequency for members, including the missing data:

```
#sorts the dataset according to most popular start and end station names
head(count(filtered_member, start_station_name, sort=T), n=10)
head(count(filtered_member, end_station_name, sort=T), n=10)
```

The results are as follows respectively:

```
         start_station_name      n
1              missing_data 119269
2        Clark St & Elm St   22633
3     Wells St & Concord Ln  17679
4       Theater on the Lake  17370
5      Broadway & Barry Ave  17309
6     Dearborn St & Erie St  17186
7  Kingsbury St & Kinzie St  17084
8     St. Clair St & Erie St  16772
9         Wells St & Elm St  16522
10      Wells St & Huron St  16113
```

```
           end_station_name      n
1              missing_data 127946
2         Clark St & Elm St  23036
3     Wells St & Concord Ln  18037
4    St. Clair St & Erie St  17852
5     Dearborn St & Erie St  17798
6      Broadway & Barry Ave  17487
7  Kingsbury St & Kinzie St  17188
8       Theater on the Lake  16860
9         Wells St & Elm St  15860
10      Wells St & Huron St  15132
```

Let's apply the same steps for casual dataset as well:

```
head(count(filtered_casual, start_station_name, sort=T), n=10)
head(count(filtered_casual, end_station_name, sort=T), n=10)
```

```
               start_station_name     n
1                    missing_data 82656
2          Streeter Dr & Grand Ave 36559
3       Lake Shore Dr & Monroe St 28233
4                 Millennium Park 24808
5             Theater on the Lake 18565
6           Michigan Ave & Oak St 18362
7      Lake Shore Dr & North Blvd 16868
8     Indiana Ave & Roosevelt Rd 15884
9          Michigan Ave & Lake St 13927
10                  Shedd Aquarium 13869
```

```
                 end_station_name      n
1                    missing_data 101092
2          Streeter Dr & Grand Ave  39507
3       Lake Shore Dr & Monroe St  27169
4                 Millennium Park  25738
5             Theater on the Lake  20801
6           Michigan Ave & Oak St  19047
7      Lake Shore Dr & North Blvd  17991
8     Indiana Ave & Roosevelt Rd  15899
9          Michigan Ave & Lake St  13328
10 Michigan Ave & Washington St  12944
```

As you see from above results, casuals tend to start and end trips from the same station while its little different for members. Also, we can't neglect missing data. I will analyze them separately later on.

Now, let's observe the most popular routes for members. For this, I created a new **routes** column, which is basically concatenation of start station and end station names with "---". Then sorted and returned most popular routes for members:

```
#concat start stn name with end stn name and observe the top 5 routes
filtered_member$routes<-paste(filtered_member$start_station_name,"---",filtered_member$end_station_name)
head(count(filtered_member, routes, sort=T), n=10)
```

```
                                                              routes     n
1                                       missing_data --- missing_data 68642
2                            Ellis Ave & 60th St --- Ellis Ave & 55th St  1409
3                             MLK Jr Dr & 29th St --- State St & 33rd St  1383
4                            Ellis Ave & 55th St --- Ellis Ave & 60th St  1316
5                             State St & 33rd St --- MLK Jr Dr & 29th St  1247
6  Lakefront Trail & Bryn Mawr Ave --- Lakefront Trail & Bryn Mawr Ave  1192
7                                   Burnham Harbor --- Burnham Harbor  1167
8                                 Montrose Harbor --- Montrose Harbor  1131
9                          Theater on the Lake --- Theater on the Lake  1123
10            Lake Shore Dr & Belmont Ave --- Lake Shore Dr & Belmont Ave  1120
```

Repeating the same step for casual dataset we get:

```
filtered_casual$routes<-paste(filtered_casual$start_station_name,"---",filtered_casual$end_station_name)
head(count(filtered_casual, routes, sort=T), n=10)
```

```
                                                        routes     n
1                                missing_data --- missing_data 49062
2          Streeter Dr & Grand Ave --- Streeter Dr & Grand Ave  8230
3      Lake Shore Dr & Monroe St --- Lake Shore Dr & Monroe St  7910
4                          Millennium Park --- Millennium Park  6248
5                Buckingham Fountain --- Buckingham Fountain  5726
6              Michigan Ave & Oak St --- Michigan Ave & Oak St  4734
7    Indiana Ave & Roosevelt Rd --- Indiana Ave & Roosevelt Rd  4272
8    Fort Dearborn Dr & 31st St --- Fort Dearborn Dr & 31st St  3870
9                Theater on the Lake --- Theater on the Lake  3616
10             Michigan Ave & 8th St --- Michigan Ave & 8th St  3562
```

We have done quite a lot of observations above. Next, I would summarize them into one table using data.table and formattable packages of R.[4] It is little pain to fill the table manually, but I think the result is worth it because everything becomes easier to understand.

```
library(data.table)
library(dplyr)
library(formattable)
library(tidyr)
df<-data.frame('User_type'=c("Member", "Casual"),
               "Amount"=c("2,352,923 (57.9%)","1,710,107 (42.1%)"),
               "Avg_and_median_trip_duration"=c("15.26 min - 11.07 min","41.83 min - 20.20 min"),
               "Avg_and_median_trip_distance"=c("2.25 km - 1.69 km", "2.19 km - 1.66 km"),
               "Busiest_day"=c("Saturday","Saturday"),
               "Preffered_bike_type"=c("docked bike","docked bike"),
               "Most_occured_route"=c("Ellis Ave & 60th St-Ellis Ave & 55th St (1,409)",
                                      "Streeter Dr & Grand Ave-Streeter Dr & Grand Ave (8,230)"))
formattable(df,
            align =c("l","c","c","c","c", "c", "r"),
            list("User_type" = formatter("span", style = ~ style(color = "grey",font.weight = "bold")) ))
```

User_type	Amount	Avg_and_median_trip_duration	Avg_and_median_trip_distance	Busiest_day	Preffered_bike_type	Most_occured_route
Member	2,352,923 (57.9%)	15.26 min - 11.07 min	2.25 km - 1.69 km	Saturday	docked bike	Ellis Ave & 60th St-Ellis Ave & 55th St (1,409)
Casual	1,710,107 (42.1%)	41.83 min - 20.20 min	2.19 km - 1.66 km	Saturday	docked bike	Streeter Dr & Grand Ave-Streeter Dr & Grand Ave (8,230)

Finally, let's work with missing data values that represents NA values. First, let's see the summary of only missing data:

```
#analyzing  missing_data values which represents NA values
missing_data<-drop_na(filter(ds, start_station_name=="missing_data"))
summary(select(missing_data, c('day_of_week', 'distance_traveled', 'duration_in_min')))
```

```
    day_of_week    distance_traveled duration_in_min
 Monday    :26686   Min.   :    0.0   Min.   :  0.000
 Tuesday   :25736   1st Qu.:  827.9   1st Qu.:  5.567
 Wednesday:26473    Median : 1518.6   Median : 10.533
 Thursday  :26719   Mean   : 2275.5   Mean   : 16.000
 Friday    :30681   3rd Qu.: 3168.9   3rd Qu.: 19.883
 Saturday  :35755   Max.   :31144.1   Max.   :480.483
 Sunday    :29875
```

Now, let's observe which day of the week and what type of bike represents missing data.

```
head(count(missing_data, member_casual, rideable_type, sort=T), n=10)
dim(filter(ds, rideable_type=='electric_bike'))
```

Results are respectively electric_bike and 888224 for the above code. Interestingly all occurence of the missing data (around 200k) of start and end station names occurred with electric bikes only. In other words, out of total 888224 electric bikes in use, around 200k has missing start or end station name.

Share

After tons of codes and analysis, it's time to share our results and to answer the question **"How can we convert casuals to members?"**.

We can't fully answer to this question and come up with a solution. Because the data given to us only shows one instance of each unique bike users. The best dataset we require is the instances of a user as casual and after becoming a member. Analyzing those observations, we could find some trend or pattern for users to convert from casual to members.

However, we still have some observations and inferences from our analysis that it's possible to come up with a possible solution. Although, it might not be effective fully. Now, let's summarize what we have observed from our analysis:

- **Member bike usage is quite similar throughout the week except Sunday, which is less than other days. We can infer that members are mostly working people that getting a membership is financially and time wise viable option.**
- **Casual usage is slow for weekdays but weekends are very popular especially Saturday.**
- **Docked bike is the most popular for both members and casuals. But we can see that casuals prefer docked bike more than members do.**
- **The average distance traveled by members and casuals are almost same, however, members average trip duration ~15 min. is almost three times less than casual mean trip duration ~42 min.**
- **Casual users tend to start and end trips from the same station while its little different for members.**
- **Most lengthy trips are taken by casuals and they are abnormally long. For instance, top five lengthy trips are 38, 37, 36, 35, 35 <u>days</u> all taken by casuals.**
- **All occurrence of the missing data (around 200k) of start and end station names occurred with electric bikes only. In other words, out of total 888224 electric bikes in use, around 200k has missing start or end station name.**

Considering the above observations and insights we can suggest the following:

We see that members take shorter trips to work with bikes during Monday to Saturday, since it is financially viable and fast transportation. However, casuals prefer longer trips especially Saturday and Sunday. Thus:

1. We could increase the renting price of the bikes for the weekend to target casual users into having a membership especially for docked bikes, since they are preferred more by causal users.
2. Providing a special service or perks for only members might motivate casual users to have a membership. Services might include free ice cream or lemonade, free tour guide, or fast line for renting without any line etc.

Also, since we know the most popular start station names and routes for casual users, we can put banners or special discount advertisements in those areas or routes that would target casual users.

Furthermore, all missing start and end station names occurred with electric bikes. We have to learn why that is the case and fix the infrastructure if necessary.

References:

[1] https://spark.apache.org/docs/latest/api/python/

[2] https://en.wikipedia.org/wiki/Haversine_formula

[3] https://gist.github.com/pavlov99/bd265be244f8a84e291e96c5656ceb5c

[4] https://www.littlemissdata.com/blog/prettytables