Divvy-Biking Beyond Boundaries

November 25, 2023

Seasons of Cycling: Analyzing Divvy's Year-Long Data Trends

1 Data Collection and Preparation

Divvy Trip Data The datasets were downloaded from this link. A total of 12 csv files (1 file per month) were upload into Python as Pandas Dataframes. The files were combined into 1 file using `.concat()` method.

[1]:
```
#import packages
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

#read each csv file
nov_22 = pd.read_csv("202211-divvy-tripdata.csv")
dec_22 = pd.read_csv("202212-divvy-tripdata.csv")
jan_23 = pd.read_csv("202301-divvy-tripdata.csv")
feb_23 = pd.read_csv("202302-divvy-tripdata.csv")
mar_23 = pd.read_csv("202303-divvy-tripdata.csv")
apr_23 = pd.read_csv("202304-divvy-tripdata.csv")
may_23 = pd.read_csv("202305-divvy-tripdata.csv")
jun_23 = pd.read_csv("202306-divvy-tripdata.csv")
jul_23 = pd.read_csv("202307-divvy-tripdata.csv")
aug_23 = pd.read_csv("202308-divvy-tripdata.csv")
sep_23 = pd.read_csv("202309-divvy-tripdata.csv")
oct_23 = pd.read_csv("202310-divvy-tripdata.csv")
print('import done')

#use concat to combine 12 csv
df=pd.
 ↪concat([nov_22,dec_22,jan_23,feb_23,mar_23,apr_23,may_23,jun_23,jul_23,aug_23,sep_23,oct_23]
 ↪ignore_index=True)

#drop unnecessary columns
#df.drop(df.columns[[5, 7]], axis=1, inplace = True)

#inspect dataframe
```

```
#df.head()
df.info()

#df.describe()
```

```
import done
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5652827 entries, 0 to 5652826
Data columns (total 13 columns):
 #   Column              Dtype  
---  ------              -----  
 0   ride_id             object 
 1   rideable_type       object 
 2   started_at          object 
 3   ended_at            object 
 4   start_station_name  object 
 5   start_station_id    object 
 6   end_station_name    object 
 7   end_station_id      object 
 8   start_lat           float64
 9   start_lng           float64
 10  end_lat             float64
 11  end_lng             float64
 12  member_casual       object 
dtypes: float64(4), object(9)
memory usage: 560.7+ MB
```

The combined dataframe has 5 million records (5,652,827) and has 13 columns (attributes). Upon closer inspection it is observed that the started_at and ended_at columns are of incorrect datatype so we have converted them back to datetime datatype using pandas to_datetime() method.

[2]:
```
df['started_at'] = pd.to_datetime(df['started_at'])
df['ended_at'] = pd.to_datetime(df['ended_at'])
```

[3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5652827 entries, 0 to 5652826
Data columns (total 13 columns):
 #   Column              Dtype         
---  ------              -----         
 0   ride_id             object        
 1   rideable_type       object        
 2   started_at          datetime64[ns]
 3   ended_at            datetime64[ns]
 4   start_station_name  object        
 5   start_station_id    object        
 6   end_station_name    object        
 7   end_station_id      object        
```

```
 8   start_lat           float64
 9   start_lng           float64
 10  end_lat             float64
 11  end_lng             float64
 12  member_casual       object
dtypes: datetime64[ns](2), float64(4), object(7)
memory usage: 560.7+ MB
```

2 Data Exploration and Cleaning

Each record in the dataframe has ride_id along with time, station information like name, location(latitude and longitude), bike type and user_type for each ride recorded. **ride_id**, Let's talk about ride_id for a sec. Think of it as the VIP pass for each record in our data party – it's unique for every guest and there are no plus-ones. The cool thing? Every ride_id is like a secret code, exactly 16 characters long. We've checked them all, and they're in perfect shape. So, guess what? We don't need to fuss over this column anymore. It's all good to go as is!

[4]:
```python
# Calculate % unique values per column
duplicates = df.nunique().reset_index()
duplicates.columns = ['column', 'unique_values']
duplicates['unique%'] = round((duplicates['unique_values'] / len(df)) * 100, 2)

# Calculate % missing values per column
missing = df.isna().sum().reset_index()
missing.columns = ['column', 'missing_values']
missing['missing%'] = round((missing['missing_values'] / len(df)) * 100, 2)

# Combine the dataframes
combined_df = pd.merge(duplicates, missing, on='column')
print(combined_df)
```

```
               column  unique_values  unique%  missing_values  missing%
0             ride_id        5652827   100.00               0      0.00
1       rideable_type              3     0.00               0      0.00
2          started_at        4764981    84.29               0      0.00
3            ended_at        4776456    84.50               0      0.00
4  start_station_name           1579     0.03          866243     15.32
5    start_station_id           1494     0.03          866375     15.33
6    end_station_name           1589     0.03          918796     16.25
7      end_station_id           1503     0.03          918937     16.26
8           start_lat         784998    13.89               0      0.00
9           start_lng         745056    13.18               0      0.00
10            end_lat          13873     0.25            6759      0.12
11            end_lng          13990     0.25            6759      0.12
12      member_casual              2     0.00               0      0.00
```

[5]:
```
#checking if all the ride_id have exactly 16 charaters
ride_id_len = (df['ride_id'].str.len()==16).all()
print(ride_id_len)

#count duplicates on unique column
# this is done to find if there are any dupicate records present in the data as␣
↪ride_id is the primary key, it is chosen
print('Total duplicates in ride_id column: ',df['ride_id'].duplicated().sum())
```

```
True
Total duplicates in ride_id column:  0
```

rideable_type, this column is like the ID badge for each bike, telling us what model was used for the ride. We've got three types listed: classic_bike, docked_bike, and electric_bike. But here's an interesting update: we realized that 'docked_bike' is actually an old name for what we now call a 'classic_bike.' So, we decided to give our data a little makeover. We've updated 'docked_bike' to 'classic_bike' across the board, and this tweak has brought a new lease of life to 86098 records. It's all about keeping things consistent and clear!

[6]:
```
#finding the unique values in rideable_type column
print(df['rideable_type'].unique())
```

```
['electric_bike' 'classic_bike' 'docked_bike']
```

[7]:
```
#counting the number of records that contained docked_bike
count_before_change = (df['rideable_type'] == 'docked_bike').sum()
print('The number of records having docked_bike type before:',␣
↪count_before_change)
#changing docked_bike to classic_bike
df['rideable_type'] = df['rideable_type'].replace('docked_bike','classic_bike')
#counting the number of records after change
count_after_change = (df['rideable_type'] == 'docked_bike').sum()
print('The number of records having docked_bike type after the change:',␣
↪count_after_change)
#checking for null values or empty values in rideable_type
print('Total number of empty values or null values in rideable_type :␣
↪',df['rideable_type'].isna().sum())
```

```
The number of records having docked_bike type before: 86098
The number of records having docked_bike type after the change: 0
Total number of empty values or null values in rideable_type :  0
```

started_at and ended_at, these columns are our timekeepers in the dataset. They don't just tell us when each bike trip kicked off and wrapped up; they're the key to unlocking much more. With these timestamps, we can calculate the duration of each ride – a vital piece of the puzzle. But there's more: by breaking down these dates and times, we can see patterns based on the day of the week and specific dates. This slicing and dicing of time not only makes our data richer for analysis but also adds a dash of life to our data visualizations. The granularity we get from these details is super valuable, helping us spot trends and derive insights that would otherwise be hidden in broader data.

[8]:
```
#we have already converted the started_at and ended_at columns to datetime␣
↪format and hence we can proceed to find the duration of the ride
df['date'] = df['started_at'].dt.date
df['day'] = df['started_at'].dt.day_name()
df['ride_duration'] = ((df['ended_at'] - df['started_at']).dt.total_seconds() /␣
↪60)
df.head()
```

[8]:
```
            ride_id  rideable_type          started_at            ended_at  \
0  BCC66FC6FAB27CC7  electric_bike 2022-11-10 06:21:55 2022-11-10 06:31:27
1  772AB67E902C180F   classic_bike 2022-11-04 07:31:55 2022-11-04 07:46:25
2  585EAD07FDEC0152   classic_bike 2022-11-21 17:20:29 2022-11-21 17:34:36
3  91C4E7ED3C262FF9   classic_bike 2022-11-25 17:29:34 2022-11-25 17:45:15
4  709206A3104CABC8   classic_bike 2022-11-29 17:24:25 2022-11-29 17:42:51

           start_station_name start_station_id        end_station_name  \
0          Canal St & Adams St            13011  St. Clair St & Erie St
1          Canal St & Adams St            13011  St. Clair St & Erie St
2  Indiana Ave & Roosevelt Rd           SL-005  St. Clair St & Erie St
3  Indiana Ave & Roosevelt Rd           SL-005  St. Clair St & Erie St
4  Indiana Ave & Roosevelt Rd           SL-005  St. Clair St & Erie St

  end_station_id  start_lat  start_lng    end_lat    end_lng member_casual  \
0          13016  41.879401 -87.639848  41.894345 -87.622798        member
1          13016  41.879255 -87.639904  41.894345 -87.622798        member
2          13016  41.867888 -87.623041  41.894345 -87.622798        member
3          13016  41.867888 -87.623041  41.894345 -87.622798        member
4          13016  41.867888 -87.623041  41.894345 -87.622798        member

         date       day  ride_duration
0  2022-11-10  Thursday       9.533333
1  2022-11-04    Friday      14.500000
2  2022-11-21    Monday      14.116667
3  2022-11-25    Friday      15.683333
4  2022-11-29   Tuesday      18.433333
```

2.0.1 Dealing with outliers

Here's a curious thing we spotted: among the sea of rides, some are as brief as under a minute, while others stretch beyond 24 hours – talk about extremes! We've decided to label these ultra-short and ultra-long rides as outliers. It's like finding a needle in a haystack, but we did it – and to keep our data neat and tidy, we're going to remove these outliers. A total of 156,036 rows, to be exact, are saying goodbye to our main dataset. But don't worry, they're not going into the data void; we're giving them a new home in a separate dataframe, df_duration_noise. This move helps us focus on the more typical rides and maintain the integrity of our analysis.

[9]:
```python
# Count rows before filtering
count_before = len(df)

# First, create a DataFrame of outliers
df_duration_noise = df[(df['ride_duration'] < 1) | (df['ride_duration'] >␣
↪24*60)]

# Then, filter df to remove outliers
df = df[(df['ride_duration'] >= 1) & (df['ride_duration'] <= 24*60)]

# Count rows after filtering
count_after = len(df)

# Print the number of rows deleted
print('This change has resulted in deleting', count_before - count_after,␣
↪'rows')

# Check the shape of df_duration_noise
df_duration_noise.shape
```

```
This change has resulted in deleting 156036 rows
```

[9]:
```
(156036, 16)
```

start_station_name and end_station_name We've noticed a bit of a puzzle: quite a few trips are missing either their starting or ending station names. Now, here's where things get interesting. For classic bikes, it's a must to have both a start and an end at a docking station. But electric bikes? They're the free spirits of our dataset – they can end their journeys pretty much anywhere, no dock required. So, to keep our data tidy and meaningful, we've made a decision: any classic bike records missing station names are going to be moved to a new home, a separate dataframe we're calling df_station_noise. This way, we keep our main dataset clean and focused on the complete journeys.

[10]:
```python
# Identify classic bike records missing station names
classic_missing_stations = (df['rideable_type'] == 'classic') &␣
↪(df['start_station_name'].isna() | df['end_station_name'].isna())

# Create df_station_noise DataFrame
df_station_noise = df[classic_missing_stations].copy()

# Update the main DataFrame by removing these records
df = df[~classic_missing_stations]
```

[11]:
```python
#storing the outliers or noise in a dataframe called df_noise, this is done to␣
↪preserve the data integrity and future analysis
df_noise = pd.concat([df_duration_noise, df_station_noise])
df_noise.head()
```

[11]:
```
                ride_id  rideable_type          started_at            ended_at  \
149  7F8CA9B17D7E2B5F   classic_bike 2022-11-03 11:52:03 2022-11-03 11:52:47
151  9320FCC9994902BC  electric_bike 2022-11-08 05:17:18 2022-11-08 05:17:21
152  E7372C2C8A9BFCA7  electric_bike 2022-11-08 05:16:41 2022-11-08 05:16:45
189  2B096F11BFFAEEF4  electric_bike 2022-11-25 10:47:39 2022-11-25 10:48:05
412  61A73ABE32A0FFE6   classic_bike 2022-11-03 15:49:17 2022-11-03 15:49:19

           start_station_name start_station_id           end_station_name  \
149  Desplaines St & Kinzie St     TA1306000003  Desplaines St & Kinzie St
151   Hoyne Ave & Balmoral Ave              655   Hoyne Ave & Balmoral Ave
152   Hoyne Ave & Balmoral Ave              655   Hoyne Ave & Balmoral Ave
189                        NaN              NaN      Ashland Ave & Lake St
412                 Walsh Park            18067                 Walsh Park

    end_station_id  start_lat  start_lng    end_lat    end_lng member_casual  \
149   TA1306000003  41.888716 -87.644448  41.888716 -87.644448        member
151            655  41.979913 -87.682015  41.979851 -87.681932        member
152            655  41.979833 -87.682010  41.979851 -87.681932        member
189          13073  41.890000 -87.670000  41.885920 -87.667170        member
412          18067  41.914610 -87.667968  41.914610 -87.667968        member

           date       day  ride_duration
149  2022-11-03  Thursday       0.733333
151  2022-11-08   Tuesday       0.050000
152  2022-11-08   Tuesday       0.066667
189  2022-11-25    Friday       0.433333
412  2022-11-03  Thursday       0.033333
```

[12]:
```python
df_noise.shape
```

[12]:
```
(156036, 16)
```

[13]:
```python
#checking member_casual column for possible values
print((df['member_casual']).unique())
df['member_casual'].isna().sum()
```

```
['member' 'casual']
```

[13]:
```
0
```

In our journey through the data, we've come across a neat little detail about the member_casual column. It turns out, it's pretty straightforward – just two types of riders here, member and casual. And guess what? There's not a single null or empty spot in sight for this column.

But here's where it gets a bit more complex: the start_station_name, start_station_id, end_station_name, and end_station_id columns are a different story. They've got a fair share of nulls and empties. However, we've decided not to show these records the exit door. Why? Because these gaps actually tell us something important – they reflect the unique flexibility of electric bikes, which don't always need a specific docking station. So, instead of dropping this valuable info, we've taken a creative turn: we're labeling these unknowns with an unknown value. This way, we acknowledge the gaps without losing the bigger picture of our bike-riding saga.

[14]:
```
print("finding the number of missing values and their percentage after removing␣
↪the outliers and noise")
missing = df.isna().sum().reset_index()
missing.columns = ['column', 'missing_values']
missing['missing%'] = round((missing['missing_values'] / len(df)) * 100, 2)
print(missing)
```

```
finding the number of missing values and their percentage after removing the
outliers and noise
                column  missing_values  missing%
0              ride_id               0      0.00
1        rideable_type               0      0.00
2           started_at               0      0.00
3             ended_at               0      0.00
4   start_station_name          821489     14.94
5     start_station_id          821614     14.95
6     end_station_name          853288     15.52
7       end_station_id          853426     15.53
8            start_lat               0      0.00
9            start_lng               0      0.00
10             end_lat             801      0.01
11             end_lng             801      0.01
12       member_casual               0      0.00
13                date               0      0.00
14                 day               0      0.00
15       ride_duration               0      0.00
```

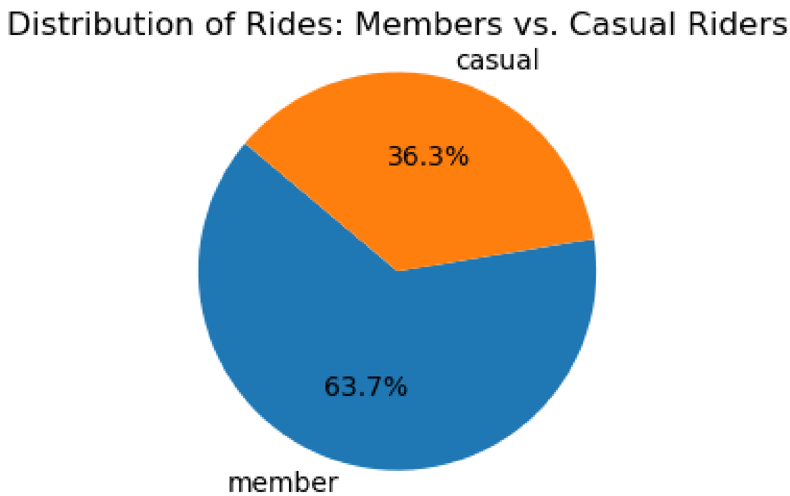
[15]:
```
# Columns to replace missing values with 'unknown'
columns_to_replace = ['start_station_name', 'start_station_id',␣
↪'end_station_name', 'end_station_id']

# Replace NaN values in each specified column with 'unknown'
for column in columns_to_replace:
    df[column] = df[column].fillna('unknown')
```

3 Data Analysis

Now moving on to the data visualization using the processed data to derive insights.

[16]:
```
ride_counts = df['member_casual'].value_counts()

# Data for the pie chart
labels = ride_counts.index
sizes = ride_counts.values
```

```python
# Plotting the pie chart
plt.figure(figsize=(4, 3))
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140)
plt.axis('equal')  # Ensures the pie chart is circular
plt.title('Distribution of Rides: Members vs. Casual Riders')
plt.show()
```



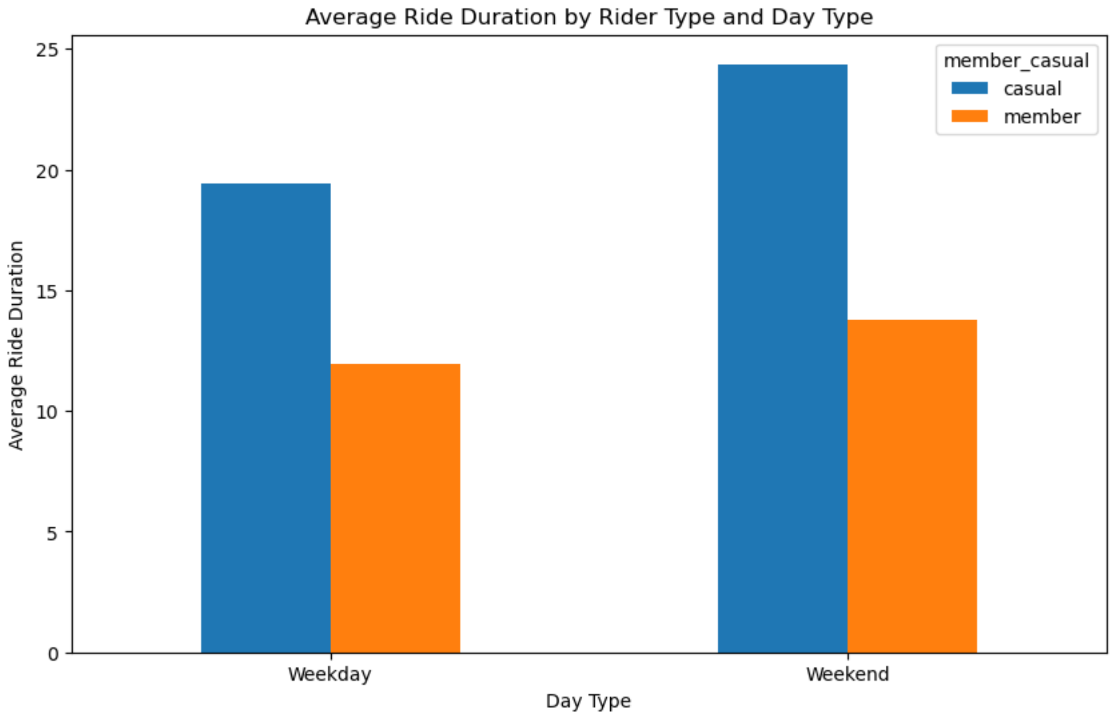
[17]:
```python
average_ride_duration = df.groupby('member_casual')['ride_duration'].mean()
print(average_ride_duration)
average_ride_duration = {'Member': 21.198452, 'Casual': 12.371359}  # in minutes
'''

# Data preparation
categories = list(average_ride_duration.keys())
values = list(average_ride_duration.values())

# Creating the bar chart
plt.figure(figsize=(8, 6))
plt.bar(categories, values, color=['blue', 'green'])
plt.xlabel('Rider Type')
plt.ylabel('Average Ride Duration (minutes)')
plt.title('Average Ride Duration: Member vs Casual Riders')
plt.xticks(categories)
plt.show()
'''
```

```
member_casual
casual     21.198452
member     12.371359
Name: ride_duration, dtype: float64
```

[17]: "\n\n# Data preparation\ncategories = list(average_ride_duration.keys())\nvalues = list(average_ride_duration.values())\n\n# Creating the bar chart\nplt.figure(figsize=(8, 6))\nplt.bar(categories, values, color=['blue', 'green'])\nplt.xlabel('Rider Type')\nplt.ylabel('Average Ride Duration (minutes)')\nplt.title('Average Ride Duration: Member vs Casual Riders')\nplt.xticks(categories)\nplt.show()\n"

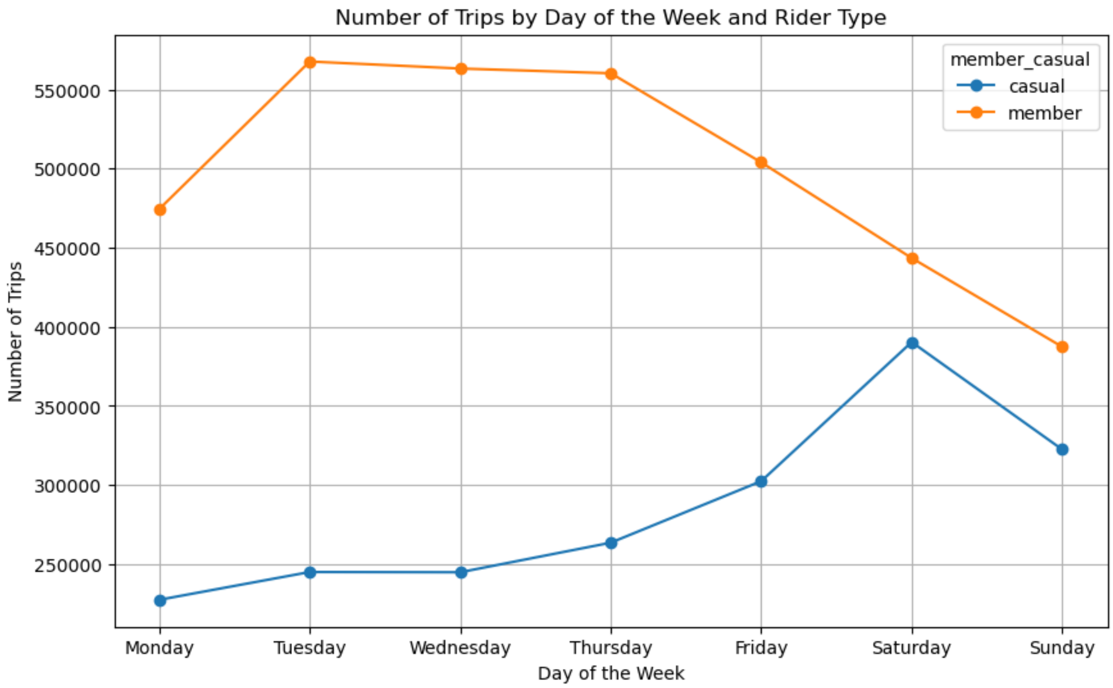
[18]:
```
# Define a function to categorize days into 'Weekday' and 'Weekend'
def categorize_day(day):
    if day.weekday() < 5:  # 0 to 4 corresponds to Monday to Friday
        return 'Weekday'
    else:  # 5 and 6 corresponds to Saturday and Sunday
        return 'Weekend'

# Apply the function to create a new column
df['day_type'] = df['date'].apply(categorize_day)

# Group by rider type and day type, then calculate the mean duration
average_duration = df.groupby(['member_casual', 'day_type'])['ride_duration'].
 ↪mean().reset_index()

# Pivot the data for easier plotting
pivot_data = average_duration.pivot(index='day_type', columns='member_casual',␣
 ↪values='ride_duration')
```

[19]:
```
# Plotting the data
pivot_data.plot(kind='bar', figsize=(10, 6))

plt.xlabel('Day Type')
plt.ylabel('Average Ride Duration')
plt.title('Average Ride Duration by Rider Type and Day Type')
plt.xticks(rotation=0)  # Rotate x-axis labels to show them horizontally
plt.show()
```



[20]:
```
df['date'] = pd.to_datetime(df['date'])
# Create a new column for the day of the week
df['day_of_week'] = df['date'].dt.day_name()

# Group by day of the week and rider type, then count the trips
trips_by_day_rider = df.groupby(['day_of_week', 'member_casual']).size().
 ↪reset_index(name='trip_count')

# Pivot the data for easier plotting
pivot_data = trips_by_day_rider.pivot(index='day_of_week',␣
 ↪columns='member_casual', values='trip_count')

# Ensure the days are ordered
days_order = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",␣
 ↪"Saturday", "Sunday"]
pivot_data = pivot_data.reindex(days_order)

# Plotting the data
pivot_data.plot(kind='line', marker='o', figsize=(10, 6))

plt.xlabel('Day of the Week')
plt.ylabel('Number of Trips')
plt.title('Number of Trips by Day of the Week and Rider Type')
```

```python
plt.grid(True)
plt.xticks(range(len(days_order)), days_order)  # Set x-ticks to days of the week
plt.show()
```



From the graph, we can infer the following:

Casual Riders:

The number of trips by casual riders is higher than that of members on every day of the week. Casual ridership appears to peak midweek, with the highest number on Wednesday, and then gradually declines towards the weekend.

Member Riders:

The pattern for member riders is quite different. The number of trips starts low on Monday, increases significantly on Tuesday, remains relatively steady through Friday, and then spikes on Saturday. The number of trips for members drops on Sunday, indicating perhaps a lesser preference for using the service on that day compared to Saturday.

Overall Trends:

Casual riders seem to use the service more consistently across the week with a peak in the middle of the week. Members show a preference for using the service towards the end of the workweek and on Saturdays.

These observations can suggest several underlying behaviors and preferences:

Casual riders might include tourists or occasional users who are more active during the week, possibly indicating leisure or errand-related activities that are not tied to the workweek schedule.

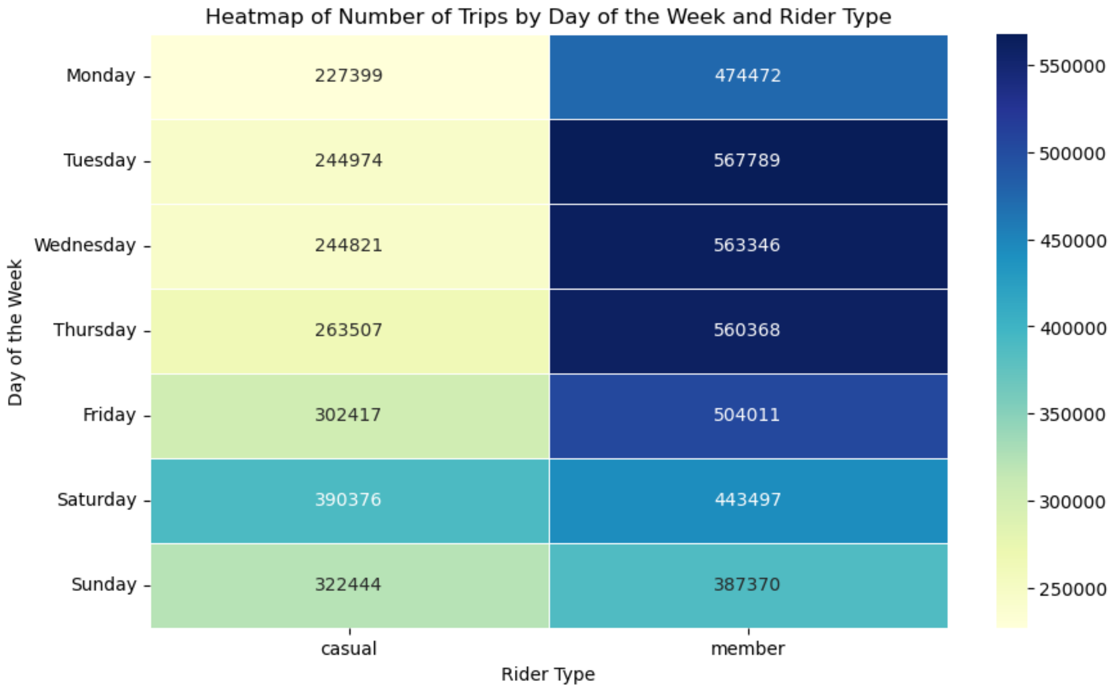
Member riders may use the service for commuting purposes, reflected in the higher number of trips on weekdays, especially towards the latter half of the week and on Saturdays for weekend activities.

The drop in member trips on Sunday might indicate a day of rest or non-reliance on bike-sharing services, possibly due to reduced work or social activities.

[21]:
```
'''
!pip install pivottablejs
from pivottablejs import pivot_ui
pivot_ui(df.head(20000))
pivot_ui(df_noise)'''
```

[21]:
```
'\n!pip install pivottablejs\nfrom pivottablejs import
pivot_ui\npivot_ui(df.head(20000))\npivot_ui(df_noise)'
```

[22]:
```
import seaborn as sns

# Creating the heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(pivot_data, annot=True, fmt="d", cmap='YlGnBu', linewidths=.5)

plt.title('Heatmap of Number of Trips by Day of the Week and Rider Type')
plt.ylabel('Day of the Week')
plt.xlabel('Rider Type')
plt.show()
```

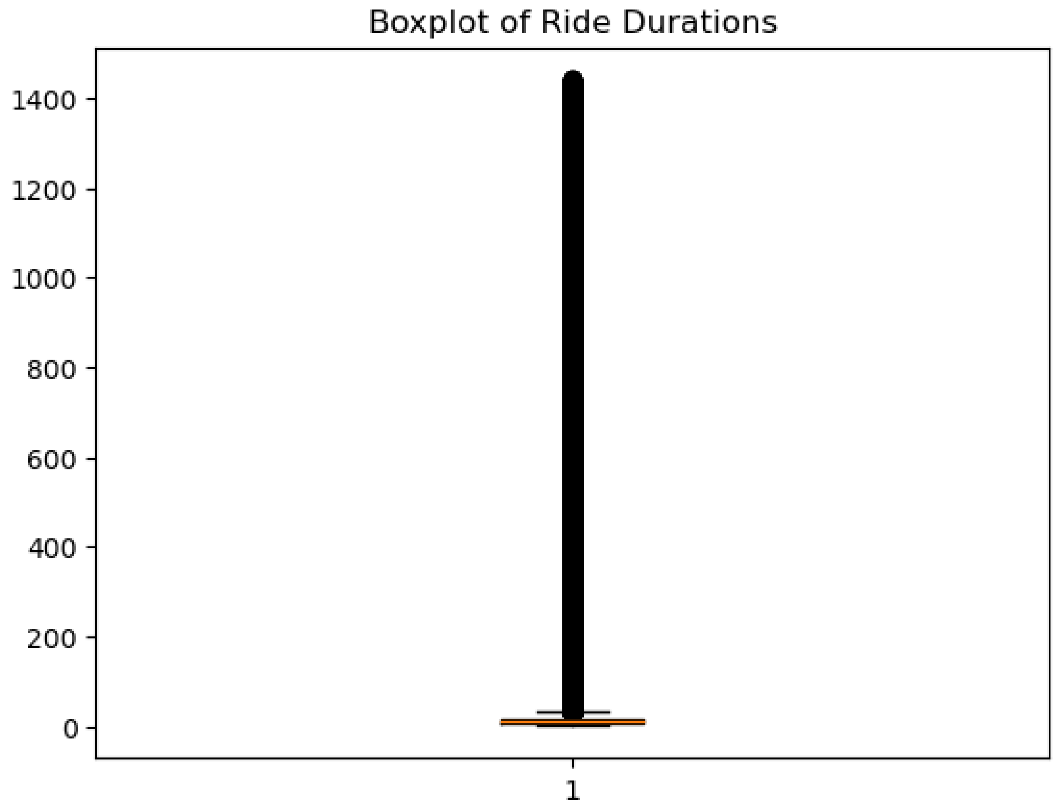


The heatmap illustrates the distribution of Divvy bike-sharing trips across different days of the week for casual and member riders. Members show a consistent increase in trips as the week progresses, peaking on Tuesday, while casual riders' trips peak on Saturdays. The heatmap's color gradient indicates that members generally take more trips than casual riders on weekdays, with the highest volume on Tuesday.

[23]:
```
import pandas as pd
from scipy import stats

# Assuming df is your DataFrame
z_scores = stats.zscore(df['ride_duration'])
outliers_z = df[(z_scores < -3) | (z_scores > 3)]
```

[24]:
```
Q1 = df['ride_duration'].quantile(0.25)
Q3 = df['ride_duration'].quantile(0.75)
IQR = Q3 - Q1

outliers_iqr = df[(df['ride_duration'] < (Q1 - 1.5 * IQR)) |␣
  ↪(df['ride_duration'] > (Q3 + 1.5 * IQR))]
```

[25]:
```
import matplotlib.pyplot as plt

plt.boxplot(df['ride_duration'])
plt.title('Boxplot of Ride Durations')
plt.show()
```



z-score outlier detection and IQR outlier detection are statistical methods used to identify abnormal points in the dataset, specifically within the ride_duration variable:

Z-Score Outlier Detection:

The z-score method standardizes the entire dataset by converting data points into z-scores, which represent the number of standard deviations a point is from the mean. This project uses a threshold of 3 standard deviations to identify outliers, meaning any ride duration more than 3 standard deviations from the mean ride duration is considered an outlier. This method is sensitive to the mean and standard deviation, and therefore can be affected by extreme values or a non-normal distribution of data.

IQR Outlier Detection:

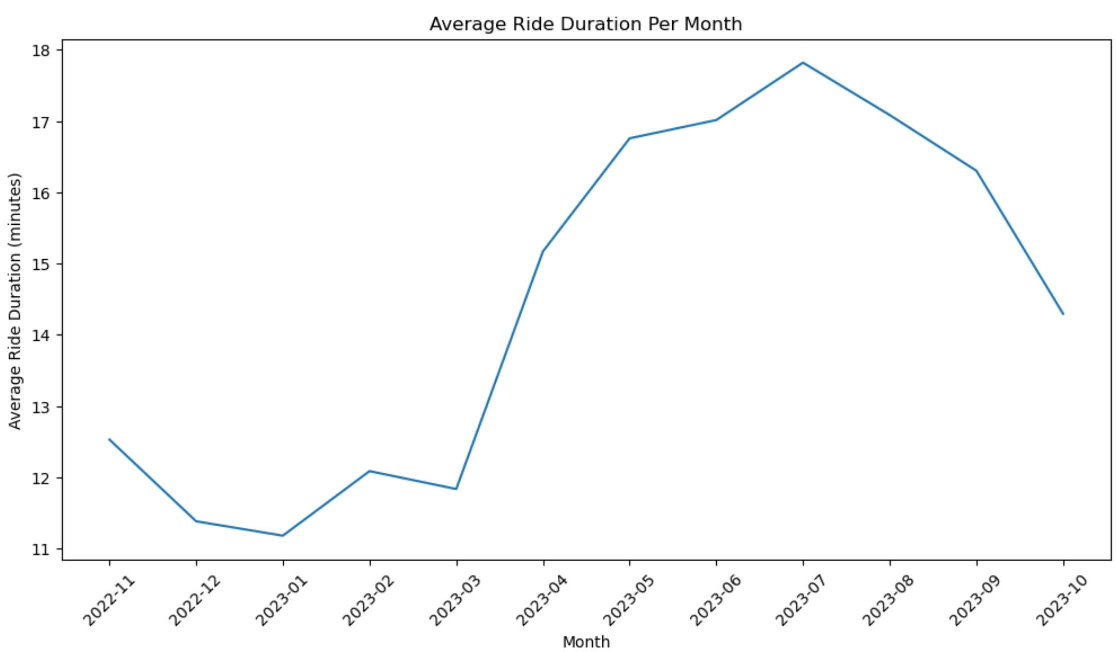
The IQR method involves calculating the interquartile range, which is the range between the first quartile (25th percentile) and the third quartile (75th percentile) of the data. Outliers are defined as observations that fall below Q1 - 1.5IQR or above Q3 + 1.5IQR. This does not assume a normal distribution of the data and is less influenced by extreme values. In this project, any ride_duration outside of these bounds is classified as an outlier and is a candidate for exclusion from further analysis to prevent skewing the results.

Boxplot Visualization:

The boxplot created in the project provides a visual representation of the distribution of ride_duration. It displays the median, quartiles, and potential outliers, which are individual points that appear outside the whiskers of the boxplot (typically set at 1.5*IQR from the quartiles). This visualization aids in confirming the presence of outliers and understanding the spread and symmetry of the data.

Both z-scores and IQR are used here to rigorously identify ride durations that are unusually long or short, which might otherwise bias the analysis. The boxplot serves as a visual confirmation of these findings, offering a clear picture of the data distribution and highlighting any potential outliers.

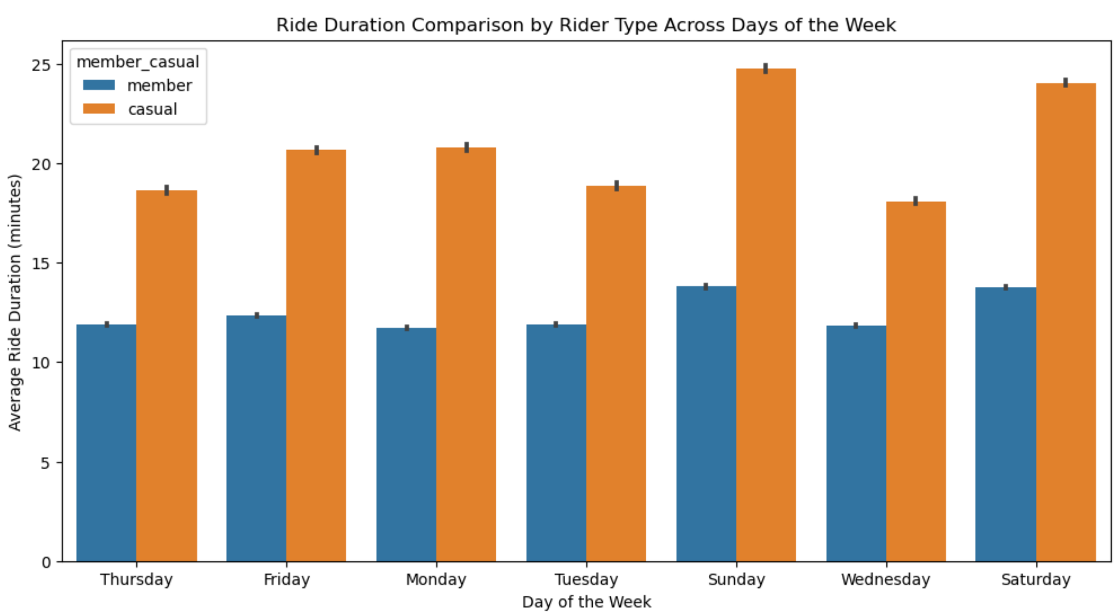
[26]:
```
#time series analysis
#This plots the average ride duration for each month to observe any trends over␣
↪time.
import matplotlib.pyplot as plt
import seaborn as sns
#df['month'] = df['started_at'].dt.to_period('M')
# Ensure 'ride_duration' is numeric
df['ride_duration'] = pd.to_numeric(df['ride_duration'], errors='coerce')

# Convert 'started_at' to Period (monthly), then to string for plotting
df['month'] = df['started_at'].dt.to_period('M').astype(str)

monthly_duration = df.groupby('month')['ride_duration'].mean().reset_index()
plt.figure(figsize=(12, 6))
sns.lineplot(data=monthly_duration, x='month', y='ride_duration')
plt.title('Average Ride Duration Per Month')
plt.xlabel('Month')
plt.ylabel('Average Ride Duration (minutes)')
plt.xticks(rotation=45)
plt.show()
```



The line graph depicts the average ride duration per month for Divvy bike-sharing from November 2022 to October 2023. There is a noticeable dip in ride durations in December 2022, after which there's a steady increase, peaking in July 2023. Following this peak, there's a sharp decline in August and September, with a slight recovery in October. This trend may suggest seasonal patterns in ride usage, with longer rides in the warmer months and shorter rides in the colder months.

The observed pattern likely reflects user behavior changes in response to weather conditions, with shorter rides during the cold winter months and longer rides during the warm summer months, indicating a preference for using bike-sharing services for longer periods when the weather is more favorable. The decline in late summer could be due to a return to school or work routines, suggesting a shift in the reasons for bike usage. The data could be vital for Divvy in planning resource allocation, and maintenance schedules to match these seasonal trends.

[27]:
```python
#rider behaviour analysis
#This will compare the ride durations between members and casual riders across␣
↪days of the week.
df['day_of_week'] = df['started_at'].dt.day_name()
plt.figure(figsize=(12, 6))
sns.barplot(data=df, x='day_of_week', y='ride_duration', hue='member_casual')
plt.title('Ride Duration Comparison by Rider Type Across Days of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Average Ride Duration (minutes)')
plt.show()
```



[28]:
```python
#Geospatial analysis
import folium
from folium.plugins import HeatMap

# Create a base map
map = folium.Map(location=[df['start_lat'].mean(), df['start_lng'].mean()],␣
↪zoom_start=12)

# Add a heatmap to the base map
HeatMap(data=df[['start_lat', 'start_lng']].dropna(), radius=10).add_to(map)

map
```

[28]: <folium.folium.Map at 0x20994e24be0>