Importing all the necessary modules

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.ticker import PercentFormatter
import seaborn as sns
```

Dropping all the unnecessary columns, working only with the relevant ones

```
df=pd.read_csv('C:/Users/Kinshuk
Mangal/Downloads/rideshare_kaggle_original.csv')
df.drop(["id",'timestamp','timezone','apparentTemperature','precipInte
nsity',
'windGust','temperatureHighTime','temperatureLowTime','apparentTempera
tureHigh',
'apparentTemperatureHighTime','apparentTemperatureLow','apparentTemper
atureLowTime',
'dewPoint','pressure','windBearing','cloudCover','uvIndex','visibility
.1',
'ozone','sunriseTime','sunsetTime','uvIndexTime','temperatureMin','tem
peratureMinTime',
'temperatureMax','temperatureMaxTime','apparentTemperatureMin','appare
ntTemperatureMinTime',
'apparentTemperatureMax','apparentTemperatureMaxTime',"windGustTime","
precipIntensityMax"]
,axis=1,inplace=True)
```

Changing the datetime to the datetime dtype

```
df.dropna(inplace=True)
df['datetime']=pd.to_datetime(df['datetime'],errors='coerce')
df.info()
```

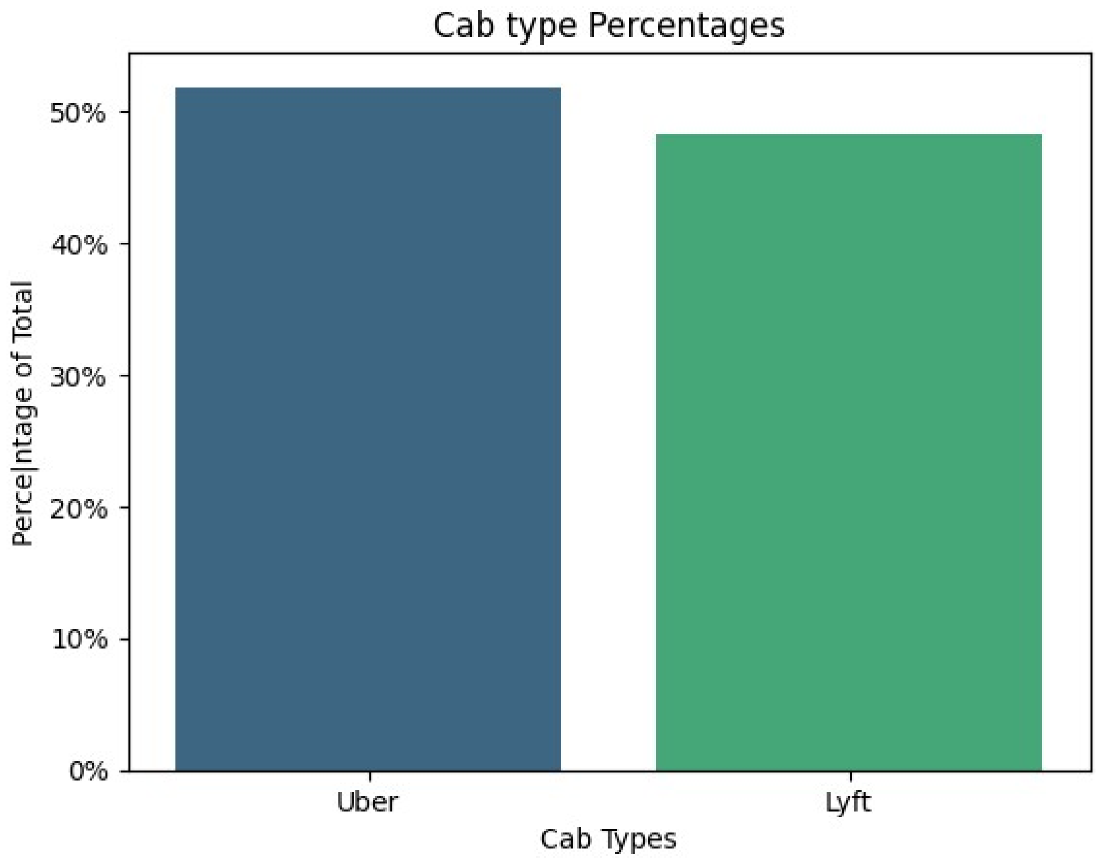
```
<class 'pandas.core.frame.DataFrame'>
Index: 637976 entries, 0 to 693070
Data columns (total 25 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   hour               637976 non-null  int64
 1   day                637976 non-null  int64
 2   month              637976 non-null  int64
 3   datetime           637976 non-null  datetime64[ns]
 4   source             637976 non-null  object
 5   destination        637976 non-null  object
 6   cab_type           637976 non-null  object
 7   product_id         637976 non-null  object
 8   name               637976 non-null  object
```

```
 9   price              637976 non-null  float64
 10  distance           637976 non-null  float64
 11  surge_multiplier   637976 non-null  float64
 12  latitude           637976 non-null  float64
 13  longitude          637976 non-null  float64
 14  temperature        637976 non-null  float64
 15  short_summary      637976 non-null  object
 16  long_summary       637976 non-null  object
 17  precipProbability  637976 non-null  float64
 18  humidity           637976 non-null  float64
 19  windSpeed          637976 non-null  float64
 20  visibility         637976 non-null  float64
 21  temperatureHigh    637976 non-null  float64
 22  temperatureLow     637976 non-null  float64
 23  icon               637976 non-null  object
 24  moonPhase          637976 non-null  float64
dtypes: datetime64[ns](1), float64(13), int64(3), object(8)
memory usage: 126.6+ MB

df.reset_index(inplace=True) #After deleting rows, resets the index to
be continuous
df.drop(['index','level_0'],axis=1,inplace=True) #Reset index adds
columns index and level_0, gettind rid of them

df.head(5)

   hour  day  month            datetime            source
destination  \
0     9   16     12 2018-12-16 09:30:07  Haymarket Square  North
Station
1     2   27     11 2018-11-27 02:00:23  Haymarket Square  North
Station
2     1   28     11 2018-11-28 01:00:22  Haymarket Square  North
Station
3     4   30     11 2018-11-30 04:53:02  Haymarket Square  North
Station
4     3   29     11 2018-11-29 03:49:20  Haymarket Square  North
Station

  cab_type    product_id          name  price  ...    short_summary  \
0     Lyft     lyft_line        Shared    5.0  ...   Mostly Cloudy
1     Lyft  lyft_premier           Lux   11.0  ...            Rain
2     Lyft          lyft          Lyft    7.0  ...           Clear
3     Lyft   lyft_luxsuv  Lux Black XL   26.0  ...           Clear
4     Lyft     lyft_plus       Lyft XL    9.0  ...   Partly Cloudy

                                        long_summary
precipProbability  \
0                           Rain throughout the day.
0.0
```

```
1   Rain until morning, starting again in the eve...
1.0
2                         Light rain in the morning.
0.0
3                  Partly cloudy throughout the day.
0.0
4                  Mostly cloudy throughout the day.
0.0

   humidity  windSpeed visibility temperatureHigh  temperatureLow  \
0      0.68       8.66     10.000           43.68           34.19
1      0.94      11.98      4.786           47.30           42.10
2      0.75       7.33     10.000           47.55           33.10
3      0.73       5.28     10.000           45.03           28.90
4      0.70       9.14     10.000           42.18           36.71

                    icon  moonPhase
0   partly-cloudy-night       0.30
1                  rain       0.64
2           clear-night       0.68
3           clear-night       0.75
4   partly-cloudy-night       0.72

[5 rows x 25 columns]
```

Cab Type Percentage

```
plt.gca().yaxis.set_major_formatter(PercentFormatter())
cabtype_percent=df['cab_type'].value_counts(normalize=True)*100
sns.barplot(x=cabtype_percent.index,y=cabtype_percent.values,
palette='viridis')
plt.xlabel("Cab Types")
plt.ylabel("Perce|ntage of Total")
plt.title("Cab type Percentages");

C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\157563769.py:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(x=cabtype_percent.index,y=cabtype_percent.values,
palette='viridis')
```

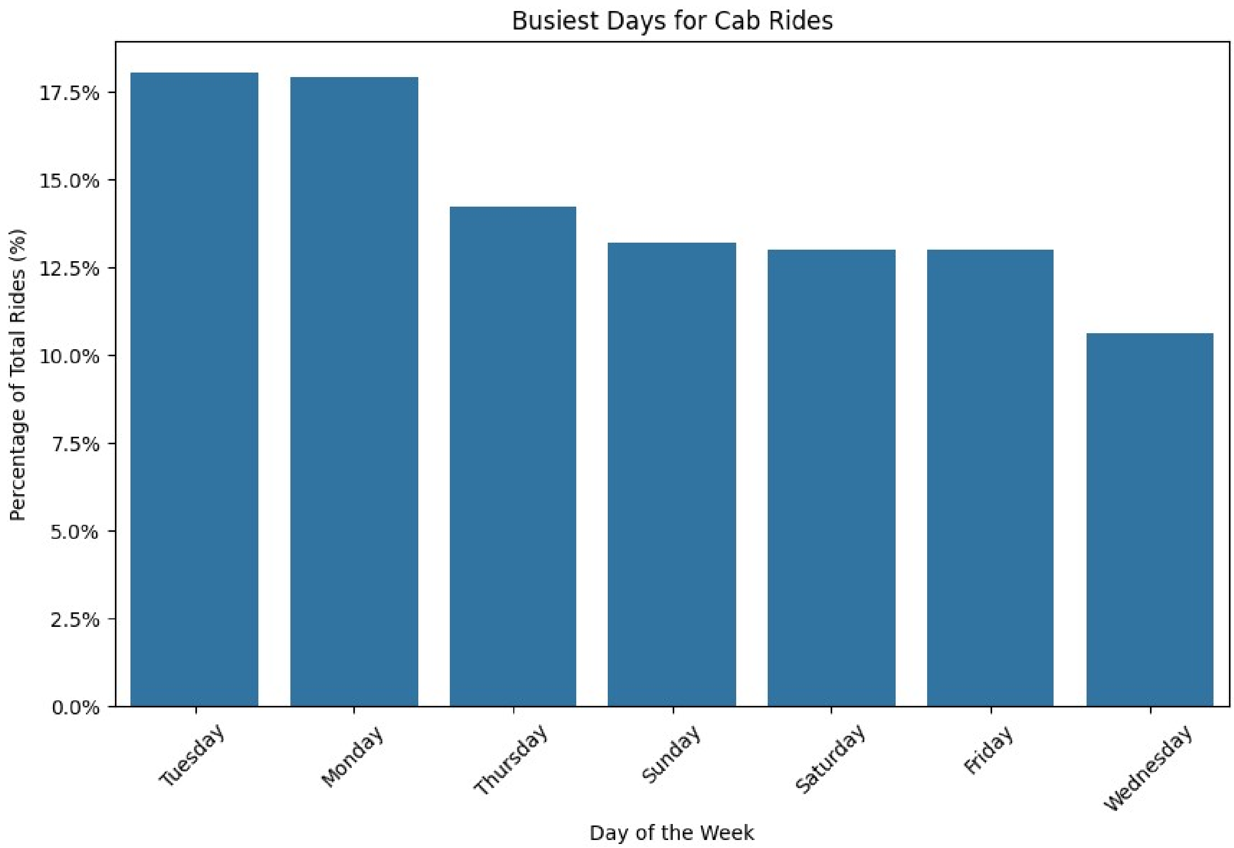


The ratio between Uber and Lyft is very similar, Uber being about 52.5% and Lyft being about 48.5%

Busiest Day of the Week

```
# 1. Extract the day of the week
df['day_of_week'] = df['datetime'].dt.day_name()

# 2. Count rides per day as percentages, sort them in descending order
rides_per_day=df['day_of_week'].value_counts(normalize=True)*100
rides_per_day=rides_per_day.reindex(['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday',
'Saturday', 'Sunday'])
rides_per_day = rides_per_day.sort_values(ascending=False)

# 3. Create the figure and apply percent formatting after initializing
the plot
plt.figure(figsize=(10,6))
sns.barplot(x=rides_per_day.index, y=rides_per_day.values)
plt.gca().yaxis.set_major_formatter(PercentFormatter())  # Correctly
set percentage format here
plt.xlabel('Day of the Week')
```

```
plt.ylabel('Percentage of Total Rides (%)')
plt.title('Busiest Days for Cab Rides')
plt.xticks(rotation=45)
plt.show()
```

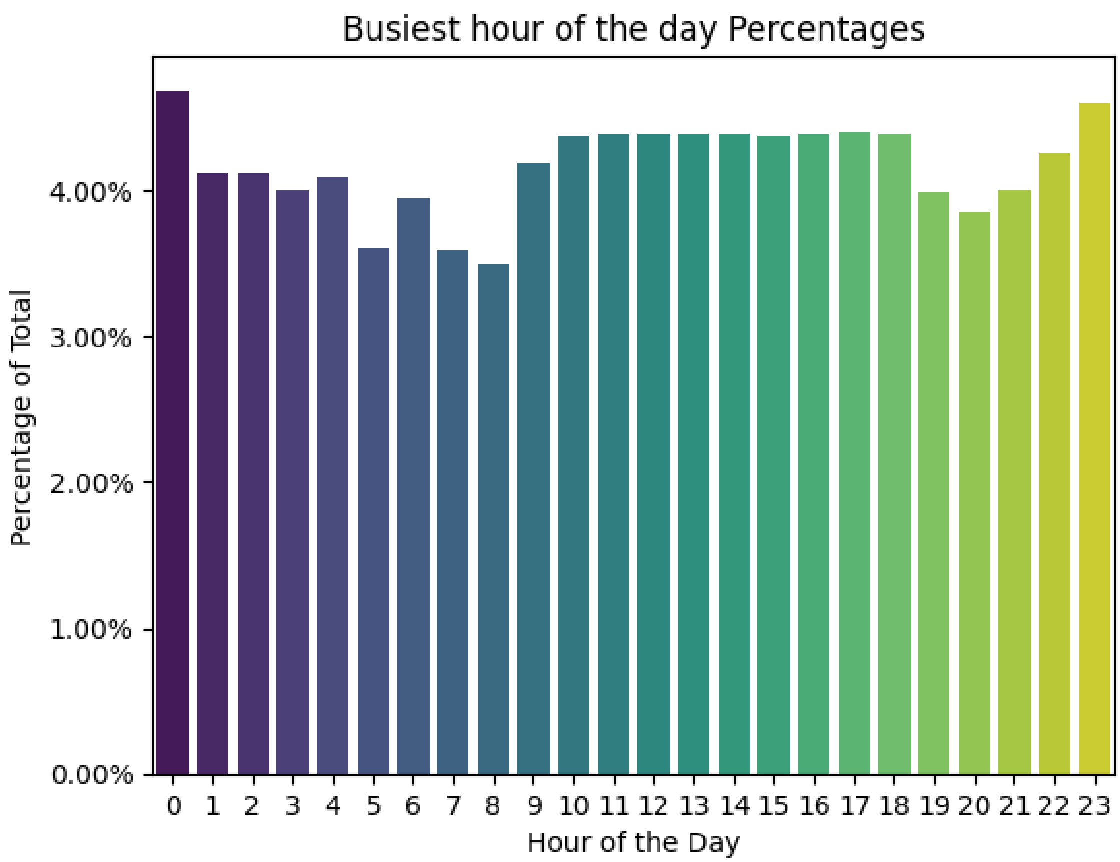


Not surprisingly, a Weekday is the busiest day of the week. The weekends are less busy in general, Wednesday is the least busiest day of the week.

Busiest hour of the Day

```
hour_of_the_day_percent=df['hour'].sort_values(ascending=False)
hour_of_the_day_percent=hour_of_the_day_percent.value_counts(normalize
=True)*100

sns.barplot(x=hour_of_the_day_percent.index,y=hour_of_the_day_percent.
values,palette='viridis')
plt.gca().yaxis.set_major_formatter(PercentFormatter())
plt.xlabel("Hour of the Day")
plt.ylabel("Percentage of Total")
plt.title("Busiest hour of the day Percentages");
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\77264989.py:4: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.


sns.barplot(x=hour_of_the_day_percent.index,y=hour_of_the_day_percent.
values,palette='viridis')
```

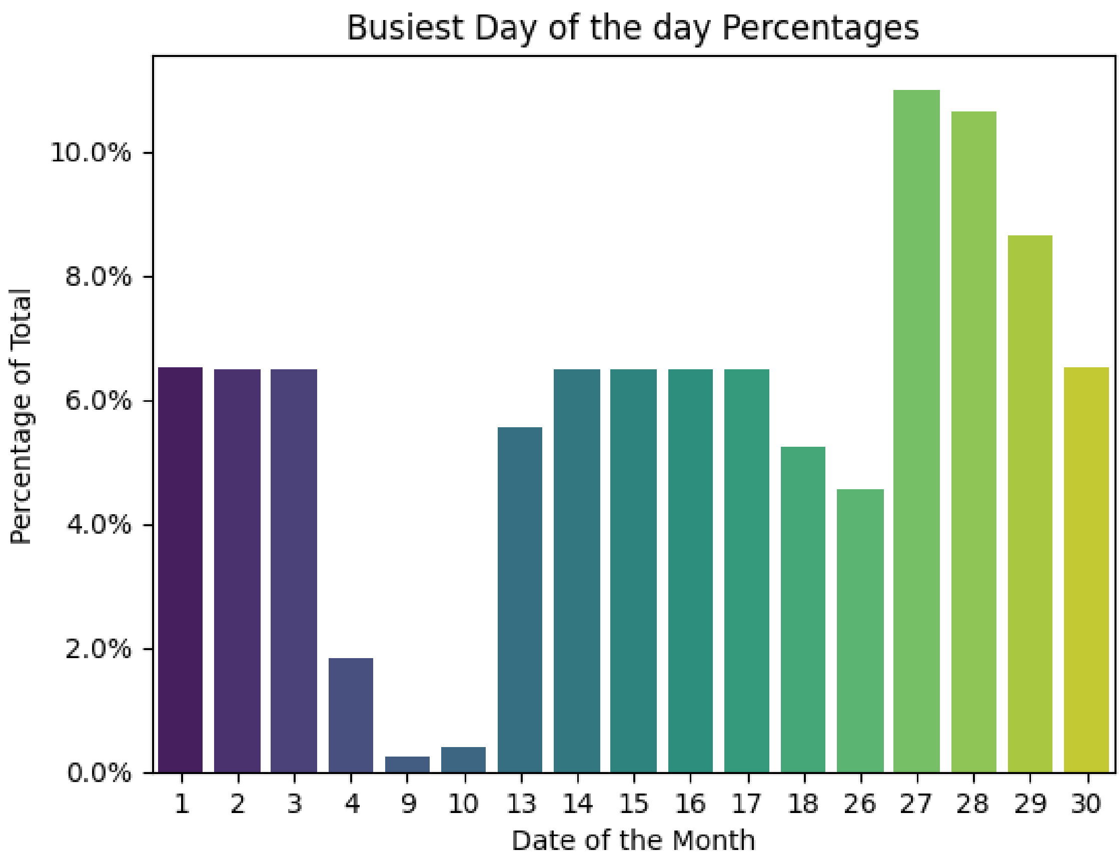


The busiest hours of the day are from 11PM-12AM. The other busy hours are from 10AM-5PM.

Busiest Date of the Month

```
date_of_the_month_percent=df['day'].value_counts(normalize=True)*100
sns.barplot(x=date_of_the_month_percent.index,y=date_of_the_month_perc
ent.values,palette='viridis')
plt.gca().yaxis.set_major_formatter(PercentFormatter())
plt.xlabel("Date of the Month")
```

```
plt.ylabel("Percentage of Total")
plt.title("Busiest Day of the day Percentages");

C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\1482125855.py:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.


sns.barplot(x=date_of_the_month_percent.index,y=date_of_the_month_perc
ent.values,palette='viridis')
```

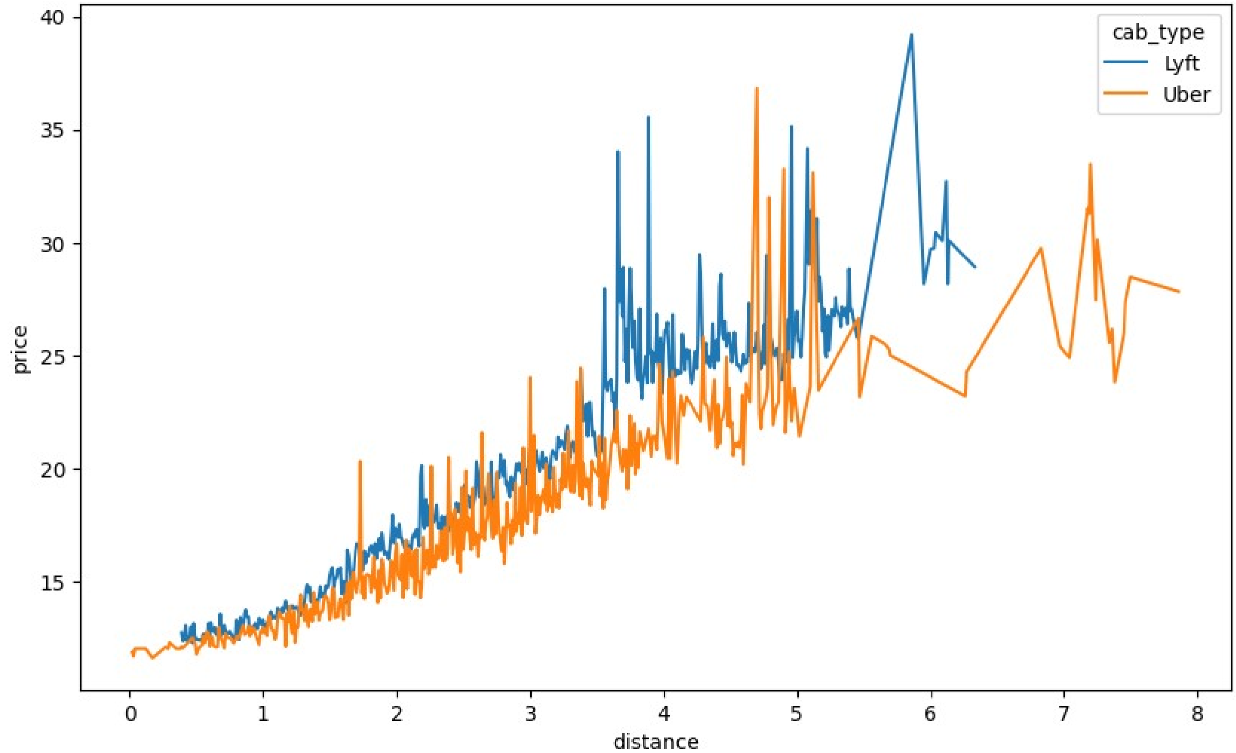


The busiest days of the dataset happen to be, not surprisingly the Black Friday, and the Weekend after that.

Average Price vs Distance

```
price_vs_distance=df.groupby(['cab_type','distance'])
['price'].mean().reset_index()
plt.figure(figsize=(10,6))
```

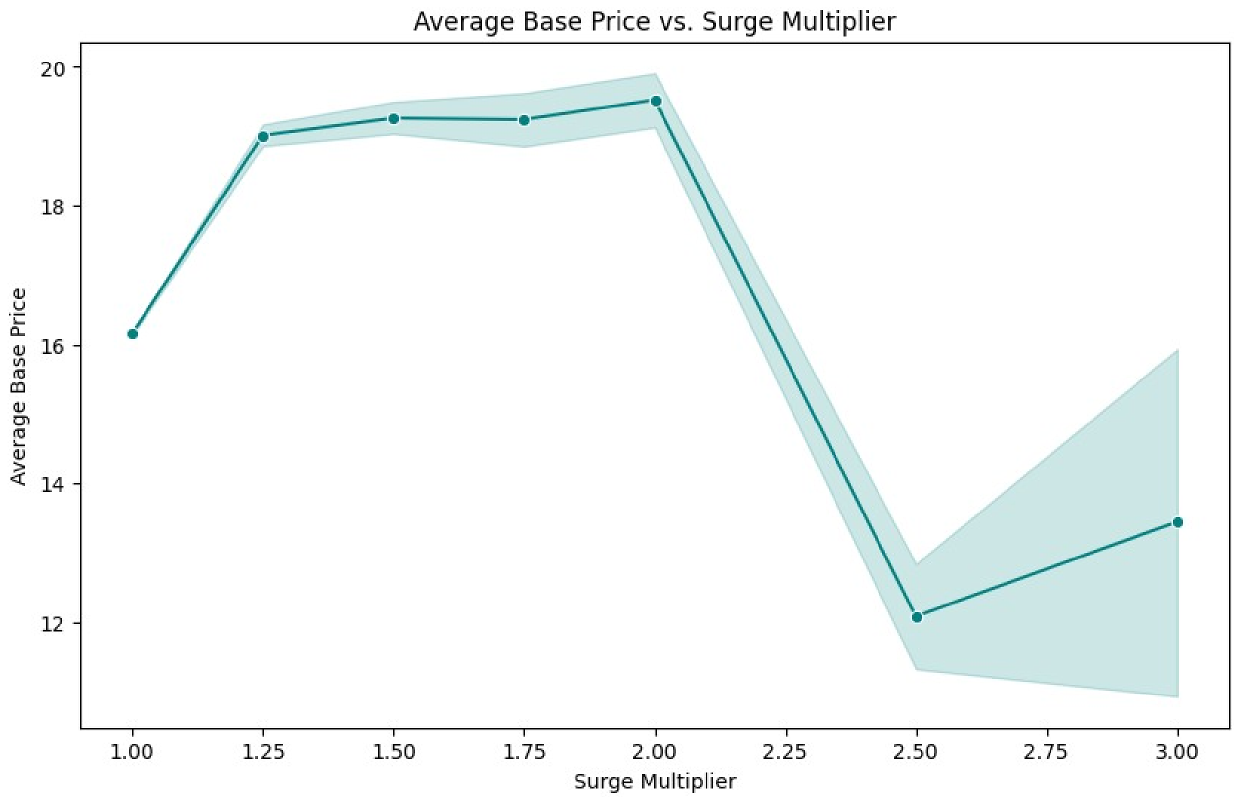
```
sns.lineplot(x='distance',y='price',hue='cab_type',data=price_vs_distance);
```



Based on this analysis, I can tell that Lyft on average costs more than Uber for the same distance. One more reason behind this could be the presence of a surge multiplier, in the Lyft Model

Base Price vs Surge Multiplier

```
df['base_price'] = df['price'] / df['surge_multiplier']

# Plot the results
plt.figure(figsize=(10, 6))
sns.lineplot(y='base_price', x='surge_multiplier', data=df,
marker='o', color='teal')

# Labeling the axes and setting the title
plt.xlabel("Surge Multiplier")
plt.ylabel("Average Base Price")
plt.title("Average Base Price vs. Surge Multiplier")
plt.show()
```

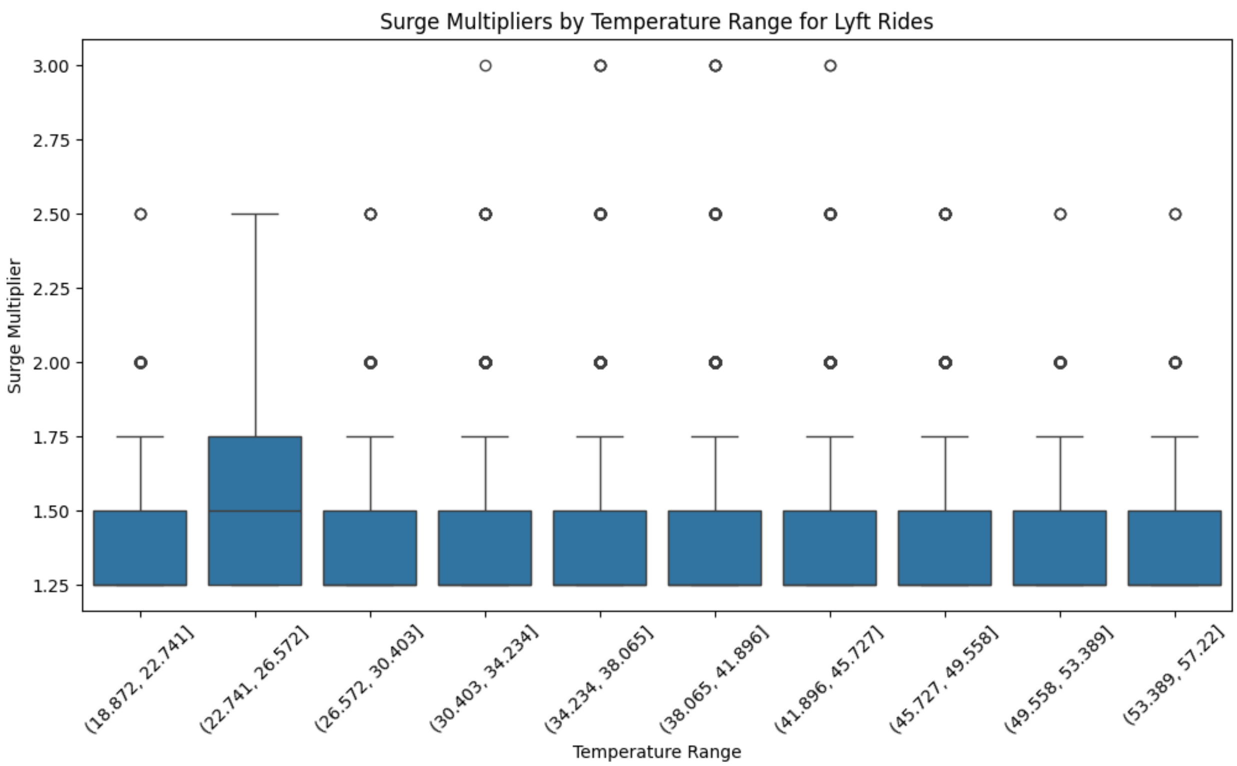


Here base price refers to the ratio of actual price over surge multiplier. The goal of this visualization was to see if the highest base prices are there for the highest surge multipliers. But it isn't necessarily true.

Surge Multiplier vs Temperature

```
# Filter for only Lyft rides and surge multipliers greater than 1
lyft_data = df[(df['cab_type'] == 'Lyft') & (df['surge_multiplier'] >
1)]

# Bin temperatures for easier visualization
temperature_bins = pd.cut(lyft_data['temperature'], bins=10)
lyft_data['temperature_bin'] = temperature_bins

# Plot surge multipliers by temperature bin
plt.figure(figsize=(12, 6))
sns.boxplot(x='temperature_bin', y='surge_multiplier', data=lyft_data)
plt.xlabel('Temperature Range')
plt.ylabel('Surge Multiplier')
plt.title('Surge Multipliers by Temperature Range for Lyft Rides')
plt.xticks(rotation=45)
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\1777968865.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  lyft_data['temperature_bin'] = temperature_bins
```



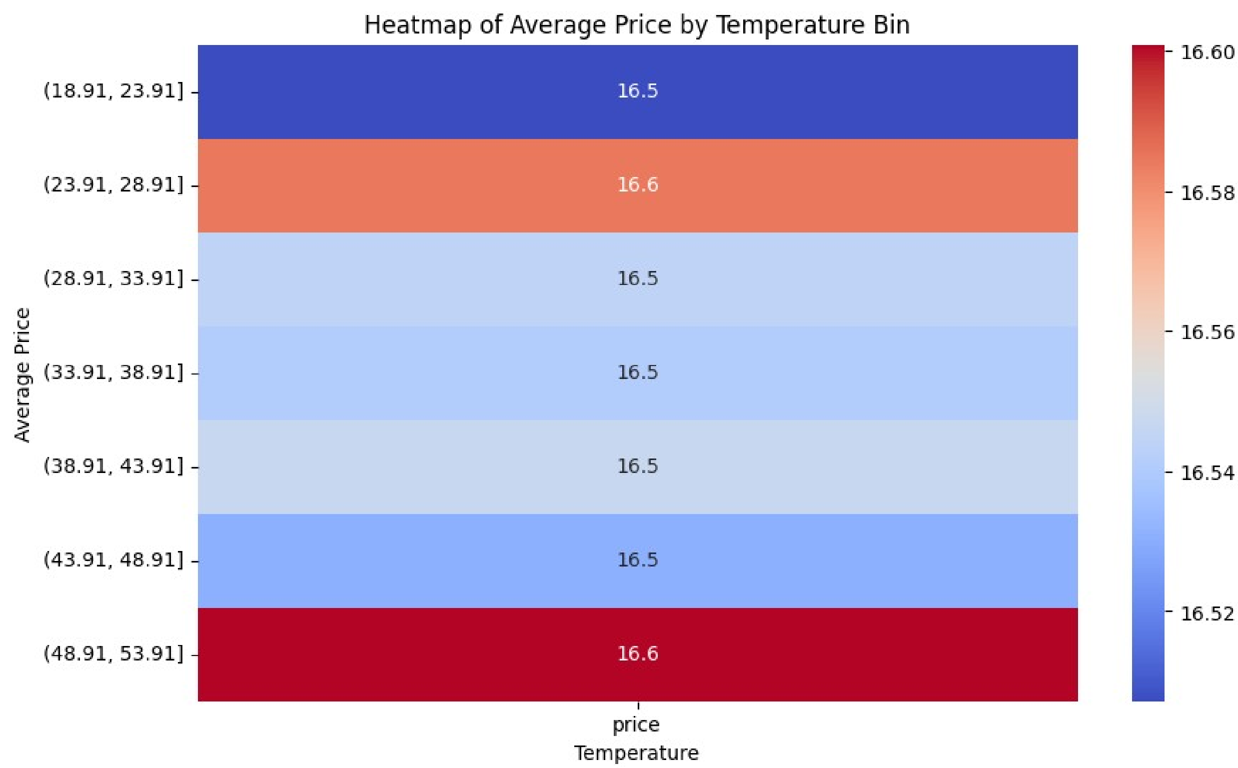
The goal was to see if there's a relationship between lower temperatures leading to higher surge multipliers, but clearly there's no indication of that.

Avg. Price vs temperature bins

```
import numpy as np
import seaborn as sns

# Create bins and calculate average price per bin
df['temp_bin'] = pd.cut(df['temperature'],
bins=np.arange(df['temperature'].min(), df['temperature'].max(), 5))
df_avg_price = df.groupby('temp_bin')['price'].mean().reset_index()

# Pivot table for heatmap
heatmap_data = df.pivot_table(index='temp_bin', values='price',
```

```
aggfunc='mean')

plt.figure(figsize=(10, 6))
sns.heatmap(heatmap_data, cmap='coolwarm', annot=True, fmt=".1f")
plt.xlabel('Temperature')
plt.ylabel('Average Price')
plt.title('Heatmap of Average Price by Temperature Bin')
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\3451997901.py:6: FutureWarning: The default of
observed=False is deprecated and will be changed to True in a future
version of pandas. Pass observed=False to retain current behavior or
observed=True to adopt the future default and silence this warning.
  df_avg_price = df.groupby('temp_bin')['price'].mean().reset_index()
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\3451997901.py:9: FutureWarning: The default value of
observed=False is deprecated and will change to observed=True in a
future version of pandas. Specify observed=False to silence this
warning and retain the current behavior
  heatmap_data = df.pivot_table(index='temp_bin', values='price',
aggfunc='mean')
```

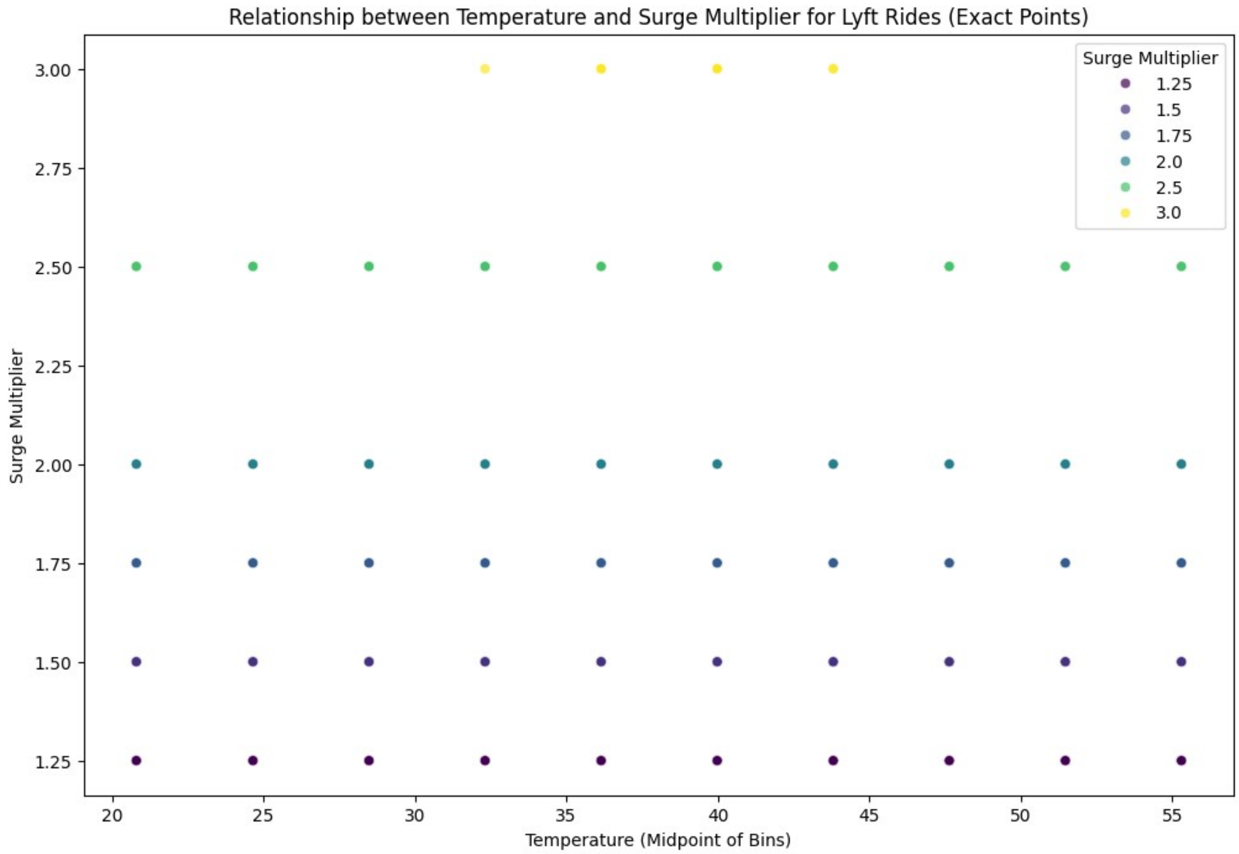


Here we look at the average price for each temperature range, we don't see any wide variations. Thus temperature doesn't seem to affect price.

Surge multiplier vs Temperature

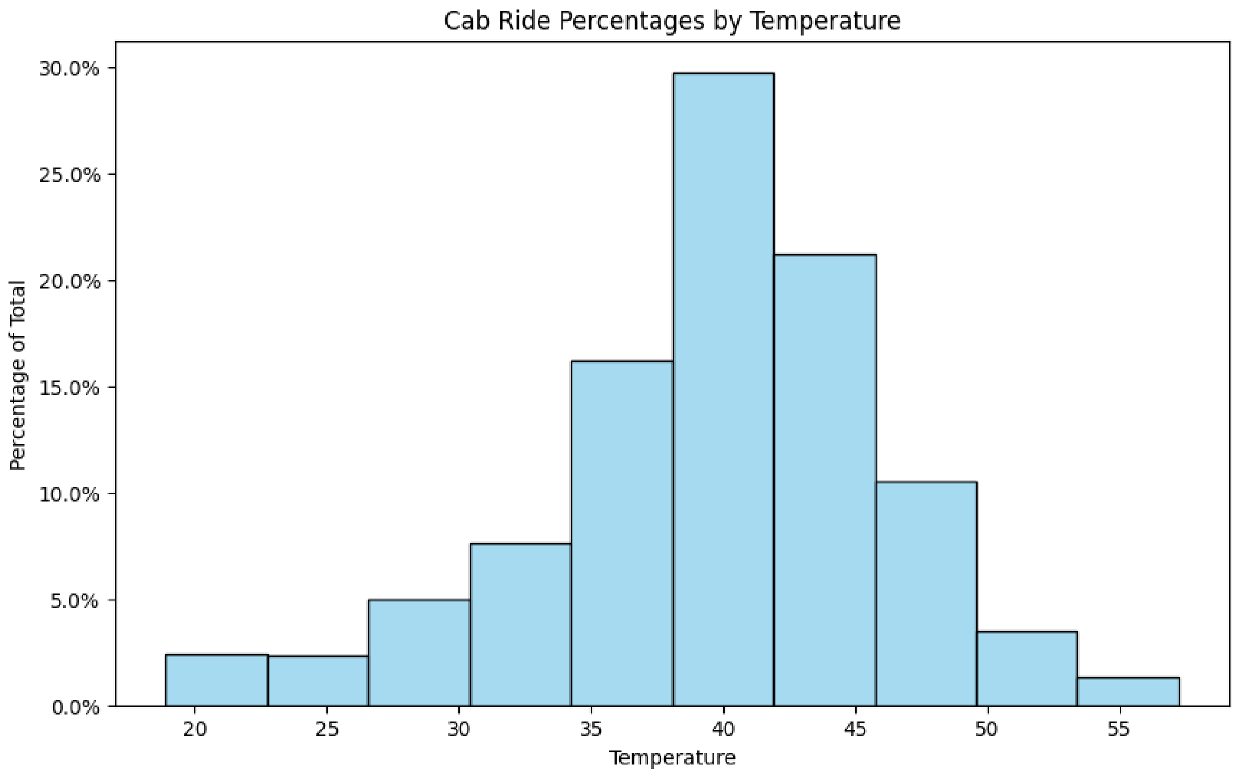
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Step 0: Filter data for Lyft rides with surge_multiplier greater
than 1
lyft_data = df[(df['cab_type'] == 'Lyft') & (df['surge_multiplier'] >
1)]

# Step 1: Bin temperatures and calculate midpoints
temperature_bins = pd.cut(lyft_data['temperature'], bins=10)
lyft_data['temperature_mid'] = temperature_bins.apply(lambda x: x.mid)

# Step 2: Plot the exact surge_multiplier points across temperature
midpoints
plt.figure(figsize=(12, 8))
sns.scatterplot(
    x='temperature_mid',
    y='surge_multiplier',
    data=lyft_data,
    hue='surge_multiplier',
    palette='viridis',
    marker='o',
    alpha=0.7
)
plt.xlabel('Temperature (Midpoint of Bins)')
plt.ylabel('Surge Multiplier')
plt.title('Relationship between Temperature and Surge Multiplier for
Lyft Rides (Exact Points)')
plt.legend(title='Surge Multiplier')
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\210212930.py:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  lyft_data['temperature_mid'] = temperature_bins.apply(lambda x:
x.mid)
```



Percentage of rides by temperature

```
# Create the histogram plot with percentages
plt.figure(figsize=(10, 6))
sns.histplot(data=df, x='temperature',
stat='percent',color='skyblue',bins=10)

# Format y-axis to show percentages
plt.gca().yaxis.set_major_formatter(PercentFormatter())

# Label the axes and set the title
plt.xlabel("Temperature")
plt.ylabel("Percentage of Total")
plt.title("Cab Ride Percentages by Temperature")
plt.show()
```

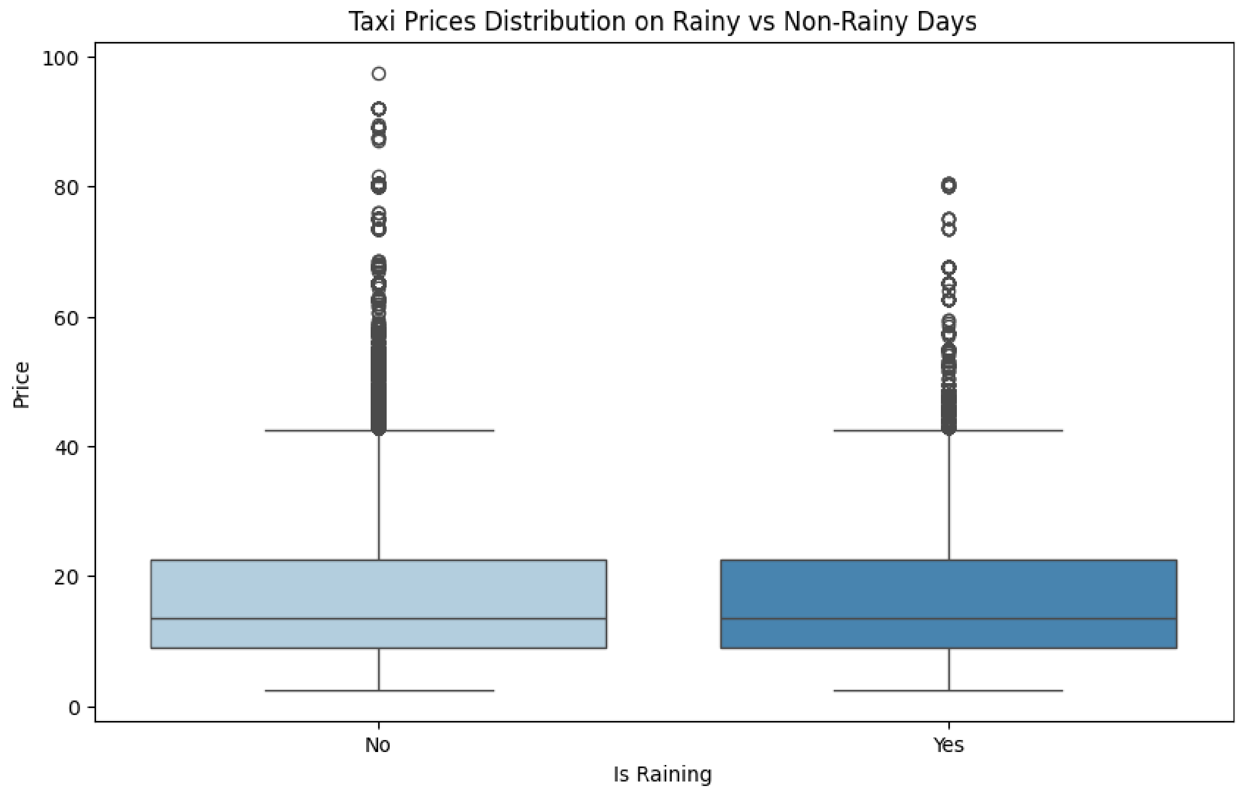


The goal was to see if a higher number of rides are taken during cold weather, but there's no such indication.

Relation b/w Price and Rain

```
# Step 1: Create a new column based on precipProbability
df['is_raining'] = df['precipProbability'] > 0.9

# Step 2: Create a box plot to visualize the distribution of prices
plt.figure(figsize=(10, 6))
sns.boxplot(x='is_raining', y='price', data=df, palette='Blues')
plt.title('Taxi Prices Distribution on Rainy vs Non-Rainy Days')
plt.xlabel('Is Raining')
plt.ylabel('Price')
plt.xticks(ticks=[0, 1], labels=['No', 'Yes'])
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\ipykernel_39840\1898327454.py:6: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.boxplot(x='is_raining', y='price', data=df, palette='Blues')
```



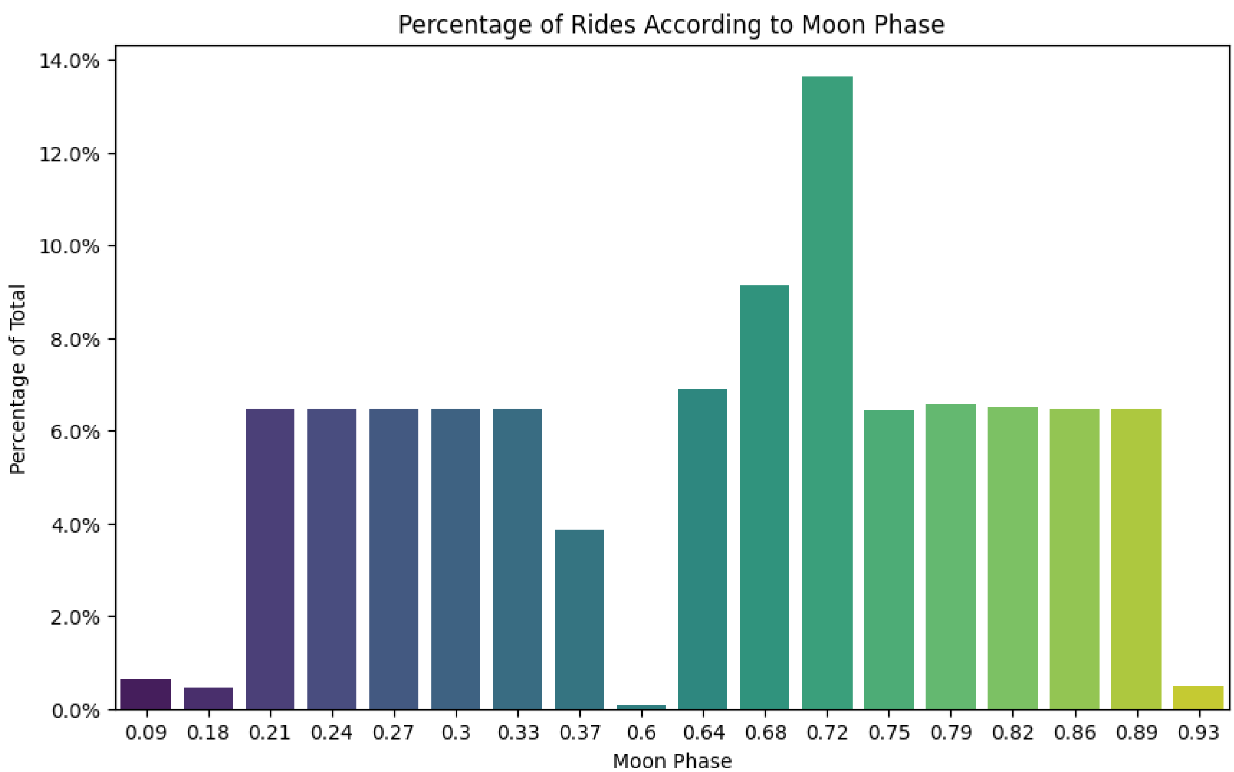
The goal was to see if rain led to higher prices or outliers, but non-rainy days have higher prices and outliers in general.

Percentage of rides for moonphase

```
from matplotlib.ticker import PercentFormatter

# Assuming `df['moonphase']` contains the moon phase data
# Step 1: Calculate percentages for each moon phase
moonphase_percent = df['moonPhase'].value_counts(normalize=True) * 100

# Step 2: Reset index to create a DataFrame for easier plotting
moonphase_percent_df = moonphase_percent.reset_index()
moonphase_percent_df.columns = ['Moon Phase', 'Percentage']

# Step 3: Create a bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x='Moon Phase', y='Percentage', data=moonphase_percent_df,
palette='viridis')

# Format the y-axis to show percentages
plt.gca().yaxis.set_major_formatter(PercentFormatter())

# Add labels and title
plt.xlabel("Moon Phase")
```

```
plt.ylabel("Percentage of Total")
plt.title("Percentage of Rides According to Moon Phase")

# Show the plot
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\2898283492.py:13: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(x='Moon Phase', y='Percentage',
data=moonphase_percent_df, palette='viridis')
```

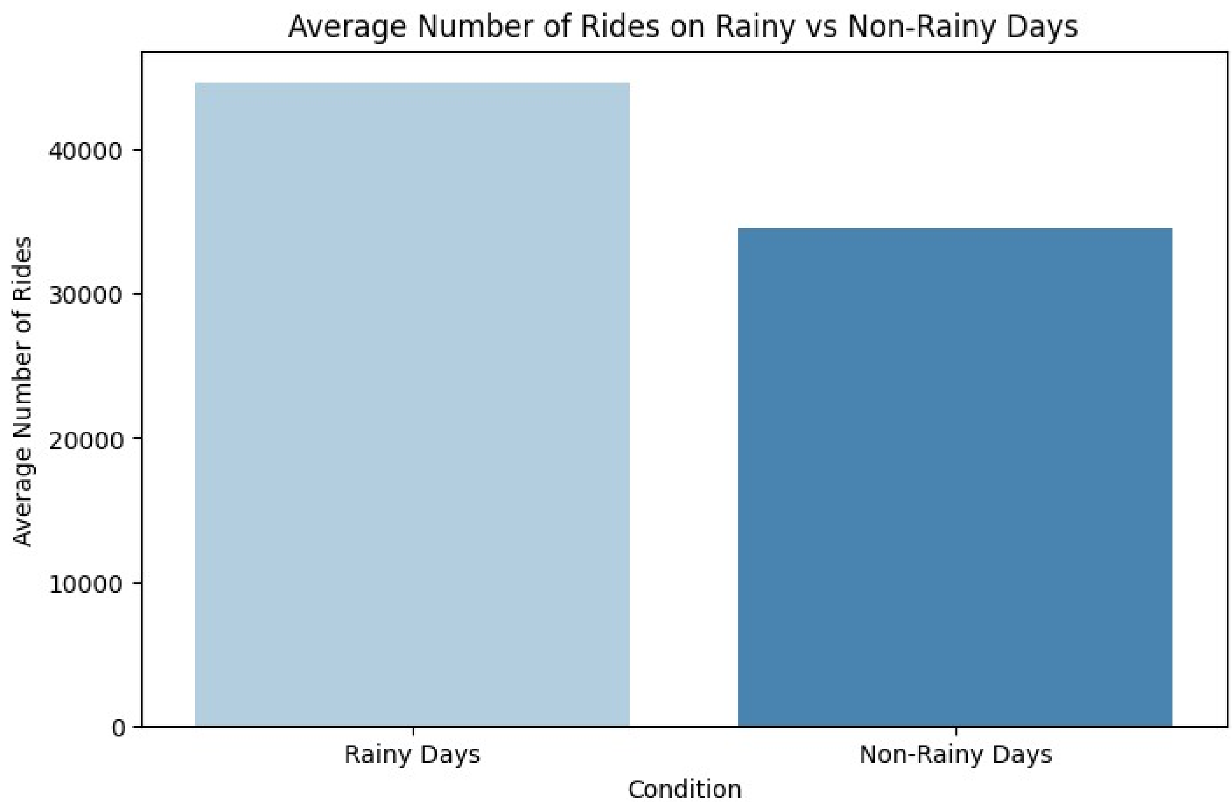


This is just a weird column present in the dataset, apparently most rides were on a waxing gibbous moonphase.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Sample DataFrame (replace this with your actual DataFrame)
# df = pd.read_csv('your_dataset.csv')  # Load your dataset
```

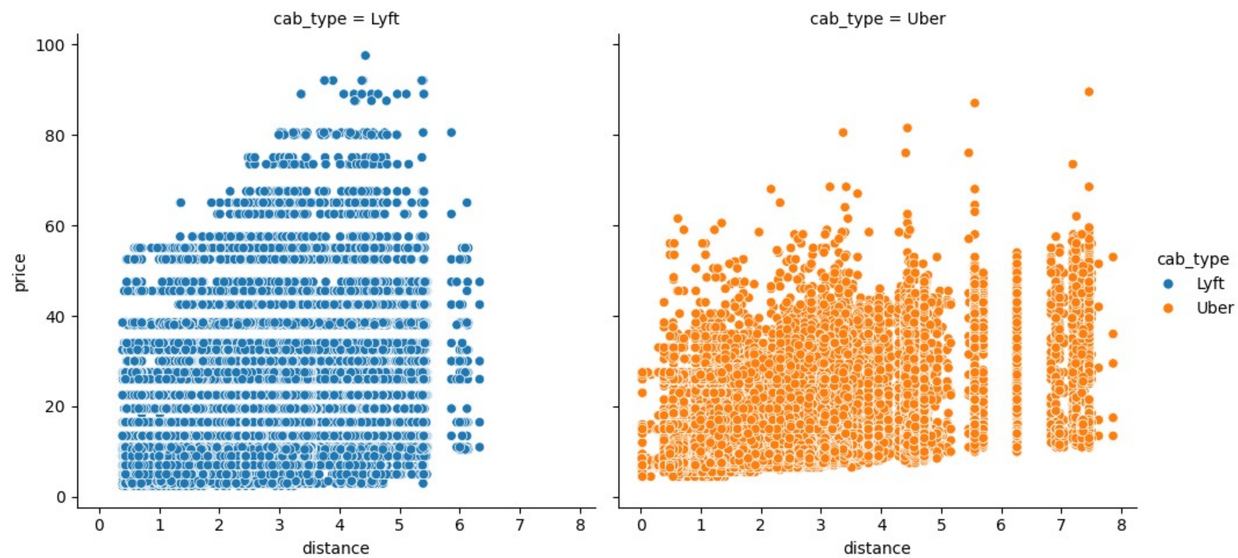
```python
# Dates when it was raining (day numbers)
rainy_dates = [27, 2, 16, 26, 17]

# Step 1: Filter the DataFrame for rainy and non-rainy days
rainy_days_data = df[df['datetime'].dt.day.isin(rainy_dates)]
non_rainy_days_data = df[~df['datetime'].dt.day.isin(rainy_dates)]

# Step 2: Calculate the number of rides for each day
rainy_rides_count =
rainy_days_data.groupby(rainy_days_data['datetime'].dt.date)
['datetime'].count().mean()
non_rainy_rides_count =
non_rainy_days_data.groupby(non_rainy_days_data['datetime'].dt.date)
['datetime'].count().mean()

# Step 3: Create a DataFrame for visualization
comparison_df = pd.DataFrame({
    'Condition': ['Rainy Days', 'Non-Rainy Days'],
    'Average Rides': [rainy_rides_count, non_rainy_rides_count]
})

# Step 4: Create a bar plot to visualize the average rides
plt.figure(figsize=(8, 5))
sns.barplot(x='Condition', y='Average Rides', data=comparison_df,
palette='Blues')
plt.title('Average Number of Rides on Rainy vs Non-Rainy Days')
plt.xlabel('Condition')
plt.ylabel('Average Number of Rides')
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_37352\4050494998.py:27: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(x='Condition', y='Average Rides', data=comparison_df,
palette='Blues')
```



Here I wanted to see if people on average took more rides in a day on a raining day, vs a non-rainy one. Clearly there were more rides on a rainy day, vs a non-rainy one.

```
plt.figure(figsize=(10, 6))
sns.relplot(x='distance', y='price', hue="cab_type", col='cab_type',
data=df, kind="scatter")
plt.show()

<Figure size 1000x600 with 0 Axes>
```

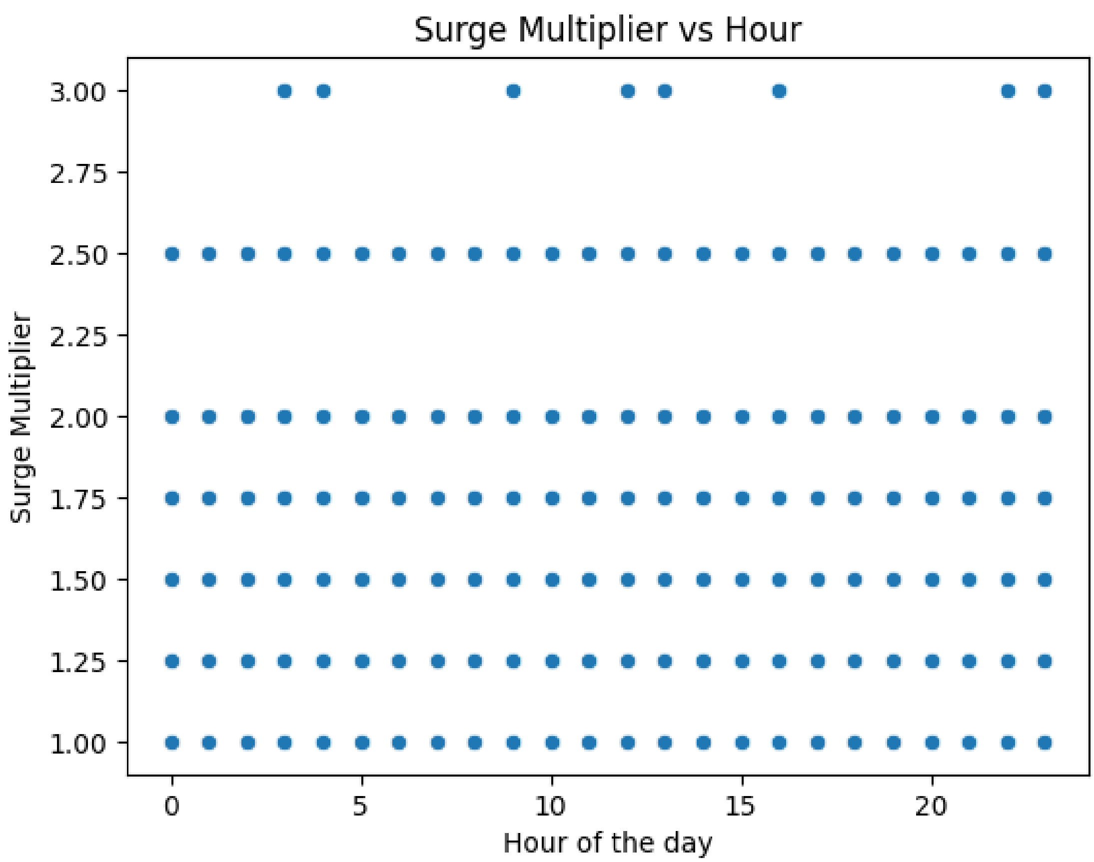


Here the goal was just to see the difference in the variation of distance vs price in Uber and Lyft separately.

Surge multiplier vs Hour

```
sns.scatterplot(x='hour',y='surge_multiplier',palette='viridis',data=d
f)
plt.xlabel("Hour of the day")
plt.ylabel("Surge Multiplier")
plt.title("Surge Multiplier vs Hour");
```

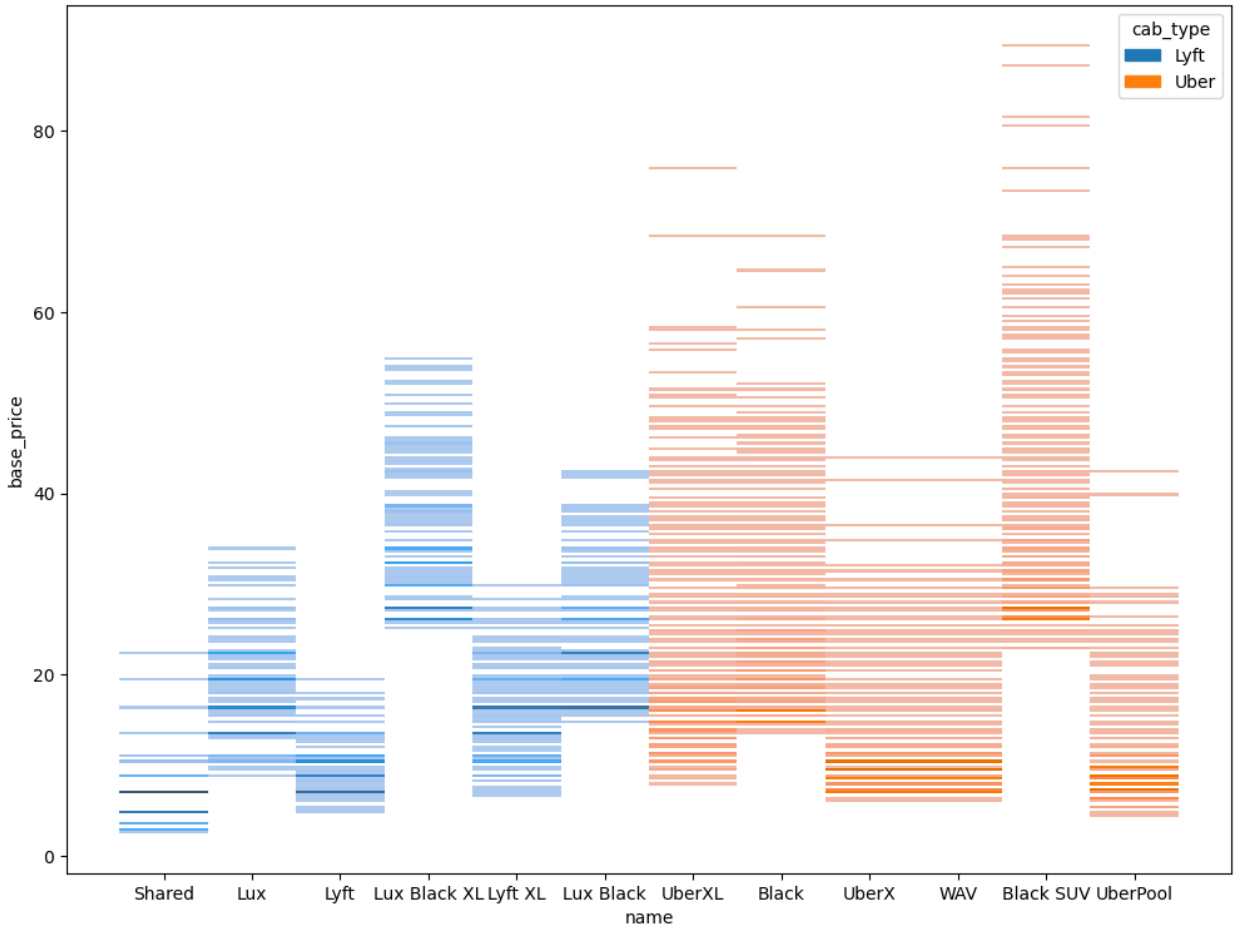
```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\380114073.py:1: UserWarning: Ignoring `palette`
because no `hue` variable has been assigned.

sns.scatterplot(x='hour',y='surge_multiplier',palette='viridis',data=d
f)
```



Here the goal was to see if there's a higher surge-multiplier in the busy hours (rush hours of the day), but this isn't neccessarily true.

Price difference between different Uber and Lyft Taxi Models

```
plt.figure(figsize=(12,9))
sns.histplot(x='name',y='base_price',data=df,hue='cab_type');
```

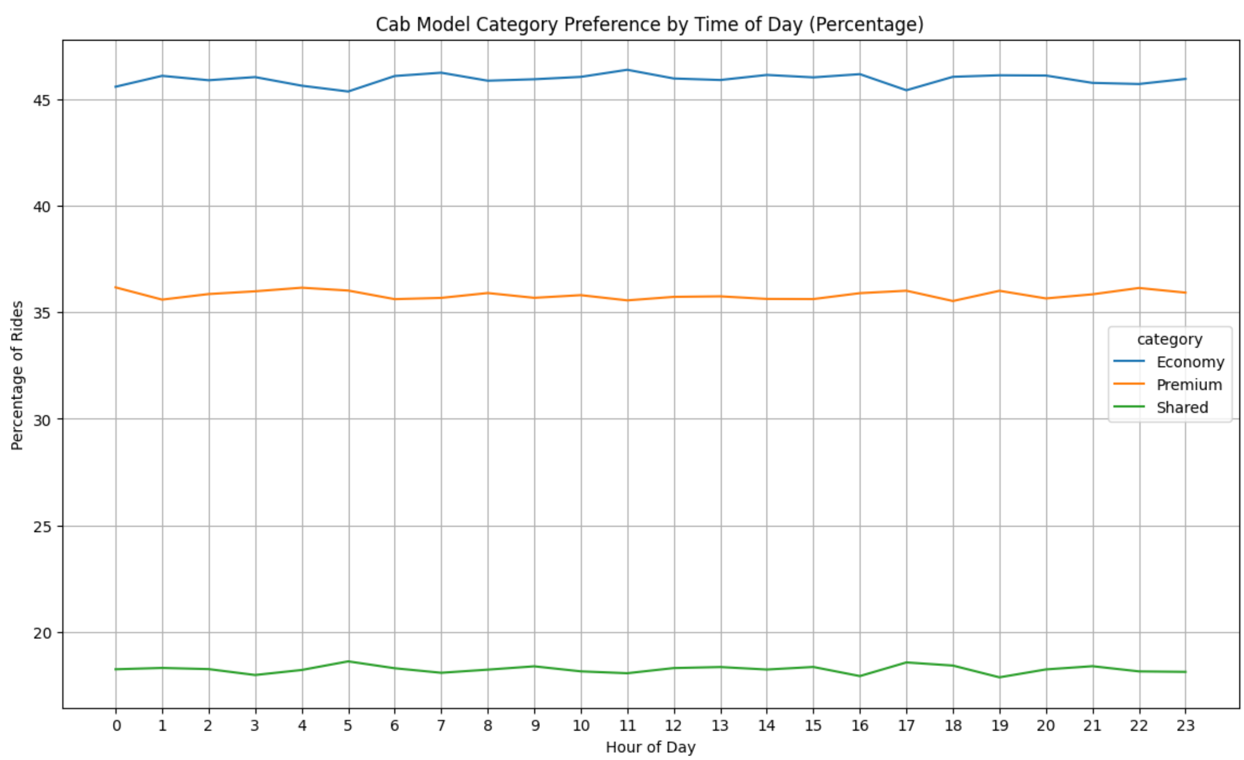


Uber and Lyft have different models, ranging from shared, economy, XL, Luxury, and XL Luxury etc. The goal was to see how they are distributed price wise on average.

Different Models during the hour of the day

```
# Create a mapping dictionary for cab models to categories
model_mapping = {
    'Lyft': 'Economy',
    'Lyft XL': 'Economy',
    'Shared': 'Shared',
    'Lux': 'Premium',
    'Lux Black': 'Premium',
    'Lux Black XL': 'Premium',
    'UberX': 'Economy',
    'UberXL': 'Economy',
    'UberPool': 'Shared',
    'Black': 'Premium',
    "WAV":"Economy"
}
```

```python
# Map the 'name' column to the broader category
df['category'] = df['name'].map(model_mapping)

# Group by hour and the new category, counting occurrences
hourly_counts = df.groupby(['hour',
'category']).size().reset_index(name='count')

# Calculate percentages for clearer comparison
hourly_counts['percentage'] = hourly_counts.groupby('hour')
['count'].transform(lambda x: (x / x.sum()) * 100)

# Plot the data as a percentage
plt.figure(figsize=(14, 8))
sns.lineplot(data=hourly_counts, x='hour', y='percentage',
hue='category')
plt.title('Cab Model Category Preference by Time of Day (Percentage)')
plt.xlabel('Hour of Day')
plt.ylabel('Percentage of Rides')
plt.xticks(range(0,24))
plt.grid(True)
plt.show()
```

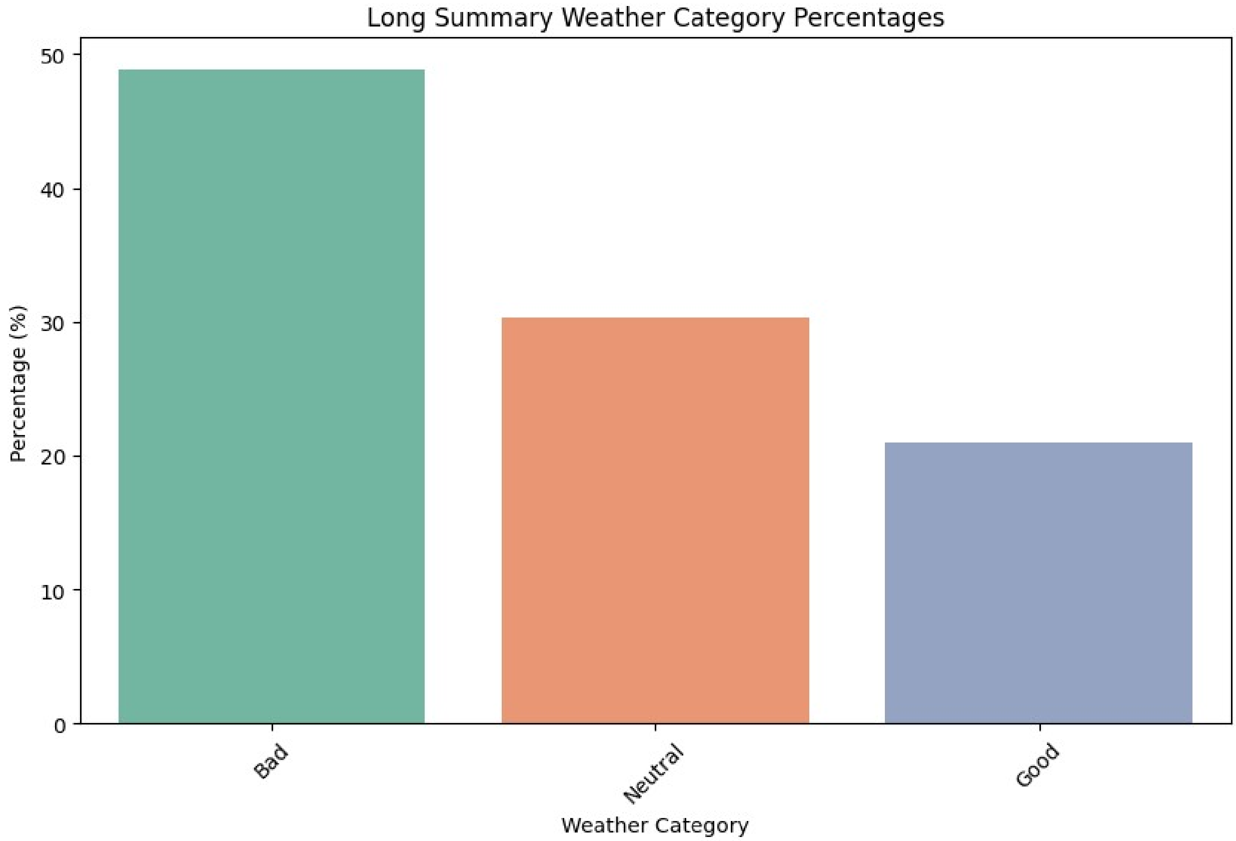


During any given time, about half the people are using the economy models, with about 37% using premium models, less than 20% use shared models.

Sentiment Analysis to price

```
# Define the mapping for weather conditions
weather_mapping = {
    'clear': 'Good',
    'sunny': 'Good',
    'partly cloudy': 'Good',
    'cloudy': 'Neutral',
    'overcast': 'Neutral',
    'foggy': 'Bad',
    'rain': 'Bad',
    'stormy': 'Bad',
    'snow': 'Bad',
    'drizzle': 'Bad'
}

# Function to categorize weather descriptions in long_summary
def categorize_weather(description):
    """Categorizes weather descriptions based on predefined
mapping."""
    if pd.isnull(description):
        return 'Unknown'
    description = description.lower()  # Normalize the description
    for key in weather_mapping.keys():
        if key in description:
            return weather_mapping[key]
    return 'Unknown'  # Return 'Unknown' if no match is found

# Apply categorization to long_summary
df['long_summary_category'] =
df['long_summary'].apply(categorize_weather)

# Display the updated DataFrame with new category
#print(df[['long_summary', 'long_summary_category']].head())

# Count the occurrences of each weather category in long_summary
long_summary_counts = df['long_summary_category'].value_counts()

# Calculate percentages
long_summary_percentages = long_summary_counts /
long_summary_counts.sum() * 100

# Visualize the percentages of long_summary categories
plt.figure(figsize=(10, 6))
sns.barplot(x=long_summary_percentages.index,
y=long_summary_percentages.values, palette='Set2')
plt.title('Long Summary Weather Category Percentages')
plt.xlabel('Weather Category')
plt.ylabel('Percentage (%)')
```

```
plt.xticks(rotation=45)
plt.show()

C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\514344079.py:40: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(x=long_summary_percentages.index,
y=long_summary_percentages.values, palette='Set2')
```



Here based on the short_summary and long_summary, the rides have been categorized into bad, neutral, and good. A majority of the rides were bad, about 30% were neutral, and a little over 20% good. This makes sense for the months of November an December.

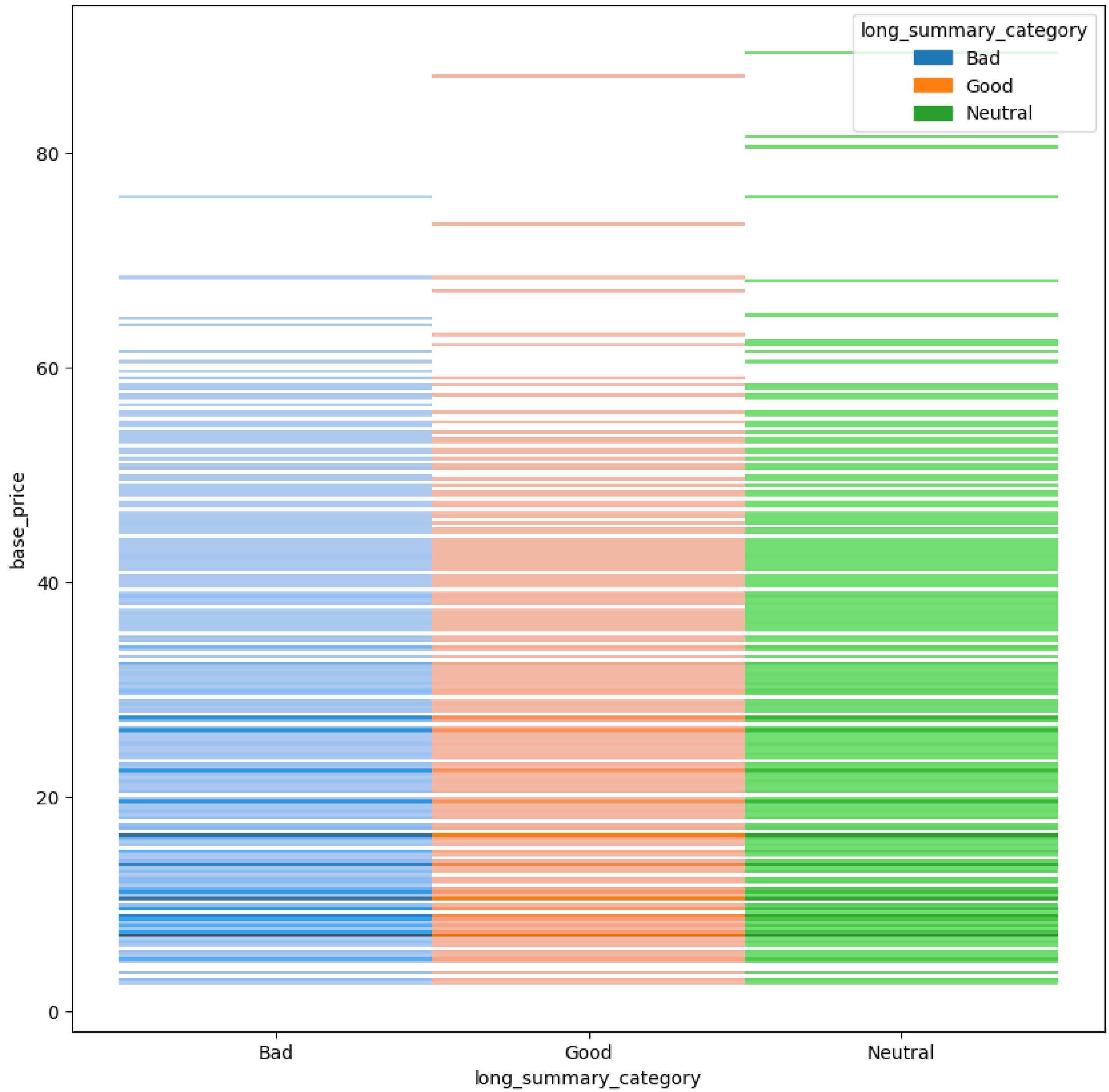
```
df.drop('short_summary_category',axis=1,inplace=True)

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call
last)
Cell In[24], line 1
```

```
----> 1 df.drop('short_summary_category',axis=1,inplace=True)

File ~\anaconda3\envs\geo_env\lib\site-packages\pandas\core\
frame.py:5581, in DataFrame.drop(self, labels, axis, index, columns,
level, inplace, errors)
   5433 def drop(
   5434     self,
   5435     labels: IndexLabel | None = None,
   (...)
   5442     errors: IgnoreRaise = "raise",
   5443 ) -> DataFrame | None:
   5444     """
   5445     Drop specified labels from rows or columns.
   5446 
   (...)
   5579             weight  1.0     0.8
   5580     """
-> 5581     return super().drop(
   5582         labels=labels,
   5583         axis=axis,
   5584         index=index,
   5585         columns=columns,
   5586         level=level,
   5587         inplace=inplace,
   5588         errors=errors,
   5589     )

File ~\anaconda3\envs\geo_env\lib\site-packages\pandas\core\
generic.py:4788, in NDFrame.drop(self, labels, axis, index, columns,
level, inplace, errors)
   4786 for axis, labels in axes.items():
   4787     if labels is not None:
-> 4788         obj = obj._drop_axis(labels, axis, level=level,
errors=errors)
   4790 if inplace:
   4791     self._update_inplace(obj)

File ~\anaconda3\envs\geo_env\lib\site-packages\pandas\core\
generic.py:4830, in NDFrame._drop_axis(self, labels, axis, level,
errors, only_slice)
   4828         new_axis = axis.drop(labels, level=level,
errors=errors)
   4829     else:
-> 4830         new_axis = axis.drop(labels, errors=errors)
   4831     indexer = axis.get_indexer(new_axis)
   4833 # Case for non-unique axis
   4834 else:

File ~\anaconda3\envs\geo_env\lib\site-packages\pandas\core\indexes\
base.py:7070, in Index.drop(self, labels, errors)
```

```
   7068 if mask.any():
   7069     if errors != "ignore":
-> 7070         raise KeyError(f"{labels[mask].tolist()} not found in
axis")
   7071     indexer = indexer[~mask]
   7072 return self.delete(indexer)

KeyError: "['short_summary_category'] not found in axis"
```

Price distribution according to Sentiment Analysis

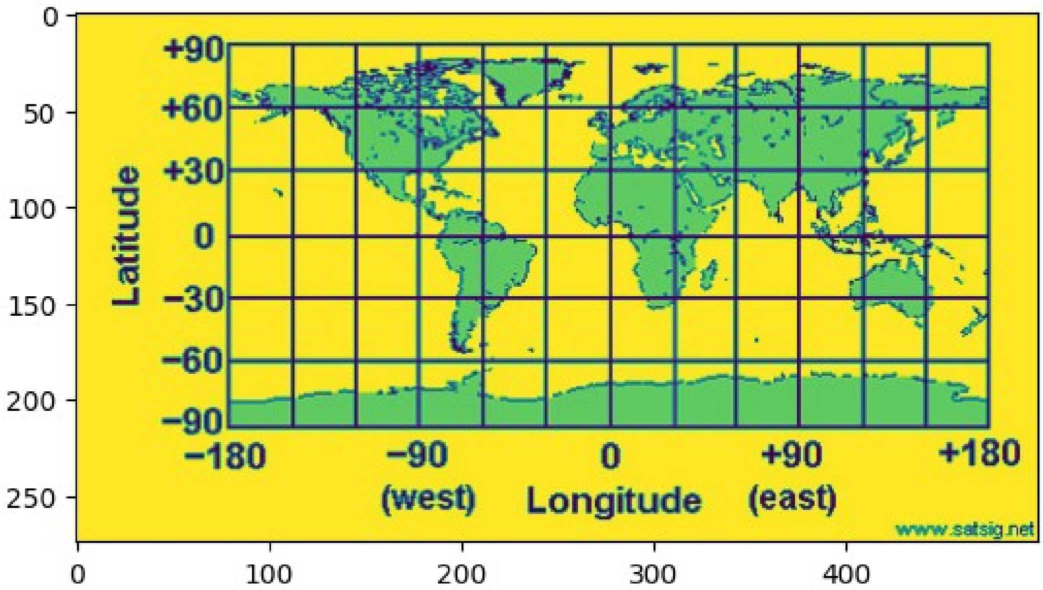
```
plt.figure(figsize=(10,10))
sns.histplot(x='long_summary_category',y='base_price',data=df,hue='lon
g_summary_category');
```



There's no evidence for bad weather rides to cost more than good or neutral.

Coordinate system for latitude and longitude

```
pic=plt.imread('geography.jpeg')
plt.imshow(pic);
```



Using API (OpenCage) to get the coordinates of the unique points in Source and Destination

```
import time

from opencage.geocoder import OpenCageGeocode

# Step 1: Extract unique source and destination locations
unique_locations = pd.concat([df['source'],
df['destination']]).unique()

# Display the unique locations
print("Unique Locations:", unique_locations)

# Initialize OpenCage Geocoder with your API key
api_key = 'a398a80e61bf47fc8ad4c2039bfceb45'  # Replace with your
OpenCage API key
geocoder = OpenCageGeocode(api_key)

# Dictionary to store coordinates
coordinates = {}

# Function to retrieve coordinates (latitude, longitude) using
OpenCage
def get_coordinates(location):
    try:
        # Query OpenCage API for the location
        result = geocoder.geocode(location)
        if result:
```

```
            # Get the first result and return latitude, longitude
            lat = result[0]['geometry']['lat']
            lon = result[0]['geometry']['lng']
            return (lat, lon)
        else:
            print(f"Could not find coordinates for {location}")
    except Exception as e:
        print(f"Error retrieving {location}: {e}")
    return None

# Step 2: Retrieve coordinates for each unique location
for location in unique_locations:
    coords = get_coordinates(location)
    if coords:
        coordinates[location] = coords
    time.sleep(1)  # Add a delay to avoid overloading the server

# Step 3: Convert the coordinates dictionary to a DataFrame for easier
merging
coords_df = pd.DataFrame.from_dict(coordinates, orient='index',
columns=['latitude', 'longitude']).reset_index()
coords_df.rename(columns={'index': 'location'}, inplace=True)

# Display the coordinates DataFrame
print(coords_df)
```

```
Unique Locations: ['Haymarket Square' 'Back Bay' 'North End' 'North
Station' 'Beacon Hill'
 'Boston University' 'Fenway' 'South Station' 'Theatre District'
 'West End' 'Financial District' 'Northeastern University']
                   location   latitude   longitude
0          Haymarket Square  41.639308  -93.696253
1                  Back Bay  42.350707  -71.079730
2                 North End  42.365097  -71.054495
3             North Station  39.466913   -0.377191
4               Beacon Hill  22.349754  114.170390
5         Boston University  42.350422  -71.103228
6                    Fenway  42.345187  -71.104599
7             South Station  42.352508  -71.054945
8          Theatre District  42.891530  -78.872486
9                  West End  55.949700   -3.213469
10       Financial District  40.707668  -74.009271
11  Northeastern University  42.338954  -71.088058
```

Here I am using an API to request the coordinates of pickup and dropoff points using a geocoding service.

```
coords_df.loc[0, 'latitude'] = 42.3617641  # Updating latitude for the
row at index 0
```

```python
coords_df.loc[0, 'longitude'] = -71.057551  # Updating longitude for the row at index 0
coords_df.loc[3, 'latitude'] = 42.366300 # Updating latitude for the row at index 2
coords_df.loc[3, 'longitude'] = -71.062220 # Updating longitude for the row at index 2
coords_df.loc[4, 'latitude'] = 42.358300 # Updating latitude for the row at index 4
coords_df.loc[4, 'longitude'] = -71.066100 # Updating longitude for the row at index 4
coords_df.loc[8, 'latitude'] = 42.353890 # Updating latitude for the row at index 8
coords_df.loc[8, 'longitude'] = -71.06278 # Updating longitude for the row at index 8
coords_df.loc[9, 'latitude'] = 42.364758 # Updating latitude for the row at index 9
coords_df.loc[9, 'longitude'] = -71.067421 # Updating longitude for the row at index 9
coords_df.loc[10, 'latitude'] = 42.355840 # Updating latitude for the row at index 10
coords_df.loc[10, 'longitude'] = -71.055620 # Updating longitude for the row at index 10
print(coords_df)
```

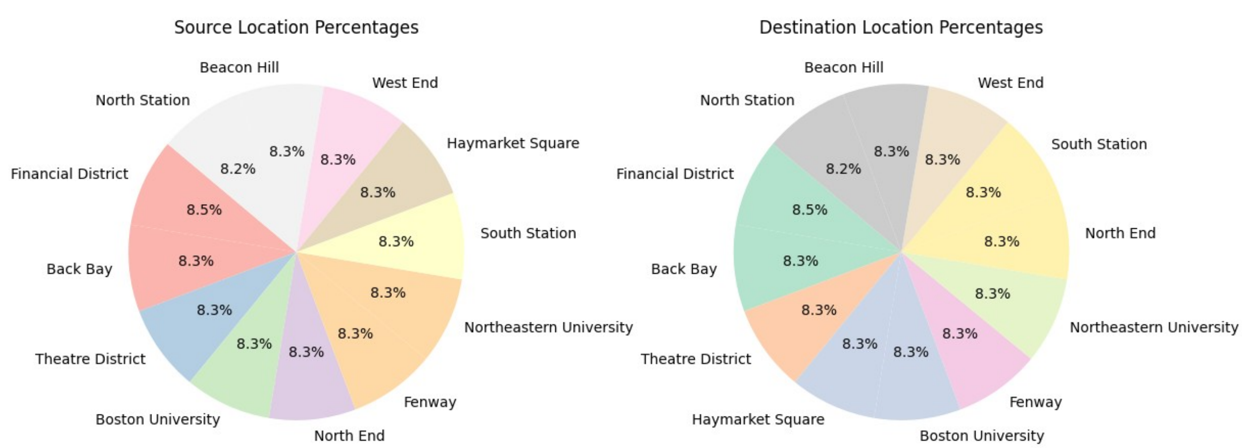
```
                   location   latitude  longitude
0           Haymarket Square  42.361764 -71.057551
1                   Back Bay  42.350707 -71.079730
2                  North End  42.365097 -71.054495
3              North Station  42.366300 -71.062220
4                Beacon Hill  42.358300 -71.066100
5          Boston University  42.350422 -71.103228
6                     Fenway  42.345187 -71.104599
7              South Station  42.352508 -71.054945
8           Theatre District  42.353890 -71.062780
9                   West End  42.364758 -71.067421
10        Financial District  42.355840 -71.055620
11  Northeastern University  42.338954 -71.088058
```

```python
#df.drop(['source_latitude','source_longitude','destination_latitude', 'destination_longitude'],axis=1,inplace=True)
coords_dict = coords_df.to_dict(orient='index')

# Step 1: Convert coords_df to separate dictionaries for latitude and longitude based on location
lat_dict = {entry['location']: entry['latitude'] for entry in coords_dict.values()}
lon_dict = {entry['location']: entry['longitude'] for entry in coords_dict.values()}

# Step 2: Map these dictionaries to add source and destination
```

```
latitude and longitude columns in df
df['source_lat'] = df['source'].map(lat_dict)
df['source_lon'] = df['source'].map(lon_dict)
df['destination_lat'] = df['destination'].map(lat_dict)
df['destination_lon'] = df['destination'].map(lon_dict)

#display(coords_dict)
```

Percentages of Source/Destination locations

```
# Calculate frequency and percentage for each unique location in
source and destination
source_counts = df['source'].value_counts(normalize=True) * 100
destination_counts = df['destination'].value_counts(normalize=True) *
100

# Plot source location percentages
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)  # First subplot
source_counts.plot(kind='pie', autopct='%1.1f%%', startangle=140,
cmap='Pastel1')
plt.title('Source Location Percentages')
plt.ylabel('')  # Hide y-axis label for clarity

# Plot destination location percentages
plt.subplot(1, 2, 2)  # Second subplot
destination_counts.plot(kind='pie', autopct='%1.1f%%', startangle=140,
cmap='Pastel2')
plt.title('Destination Location Percentages')
plt.ylabel('')  # Hide y-axis label for clarity

plt.tight_layout()
plt.show()
```



The pie chart shwows that the divide of pickup and dropoff locations is pretty similar.

Visualization of these locations on a map with Boston City outlines

```
import folium
import geopandas as gpd

# Define your location data as a dictionary
locations = {
    'location': ['Haymarket Square', 'Back Bay', 'North End', 'North
Station', 'Beacon Hill',
                 'Boston University', 'Fenway', 'South Station',
'Theatre District', 'West End',
                 'Financial District', 'Northeastern University'],
    'latitude': [42.3617641, 42.3507067, 42.3650974, 42.3663, 42.3583,
                 42.3504215, 42.3451868, 42.3525085, 42.35389,
42.364758,
                 42.35584, 42.3389545],
    'longitude': [-71.057551, -71.0797297, -71.0544954, -71.06222, -
71.0661,
                  -71.1032283, -71.1045987, -71.0549447, -71.06278, -
71.067421,
                  -71.05562, -71.088058]
}

# Load Boston boundary GeoJSON file for map overlay
boston_boundary =
gpd.read_file("City_of_Boston_Outline_Boundary_(Water_Excluded).geojso
n")

# Initialize a Folium map centered on Boston
m = folium.Map(location=[42.3601, -71.0589], zoom_start=13)

# Add Boston boundary to the map
folium.GeoJson(boston_boundary).add_to(m)

# Add markers with pop-ups for each location
for loc in locations['location']:
    lat = locations['latitude'][locations['location'].index(loc)]
    lon = locations['longitude'][locations['location'].index(loc)]
    folium.Marker(
        location=[lat, lon],
        popup=loc,
        icon=folium.Icon(color="blue", icon="info-sign")
    ).add_to(m)

# Display the map
```

```
m.save("boston_map.html")
m
```

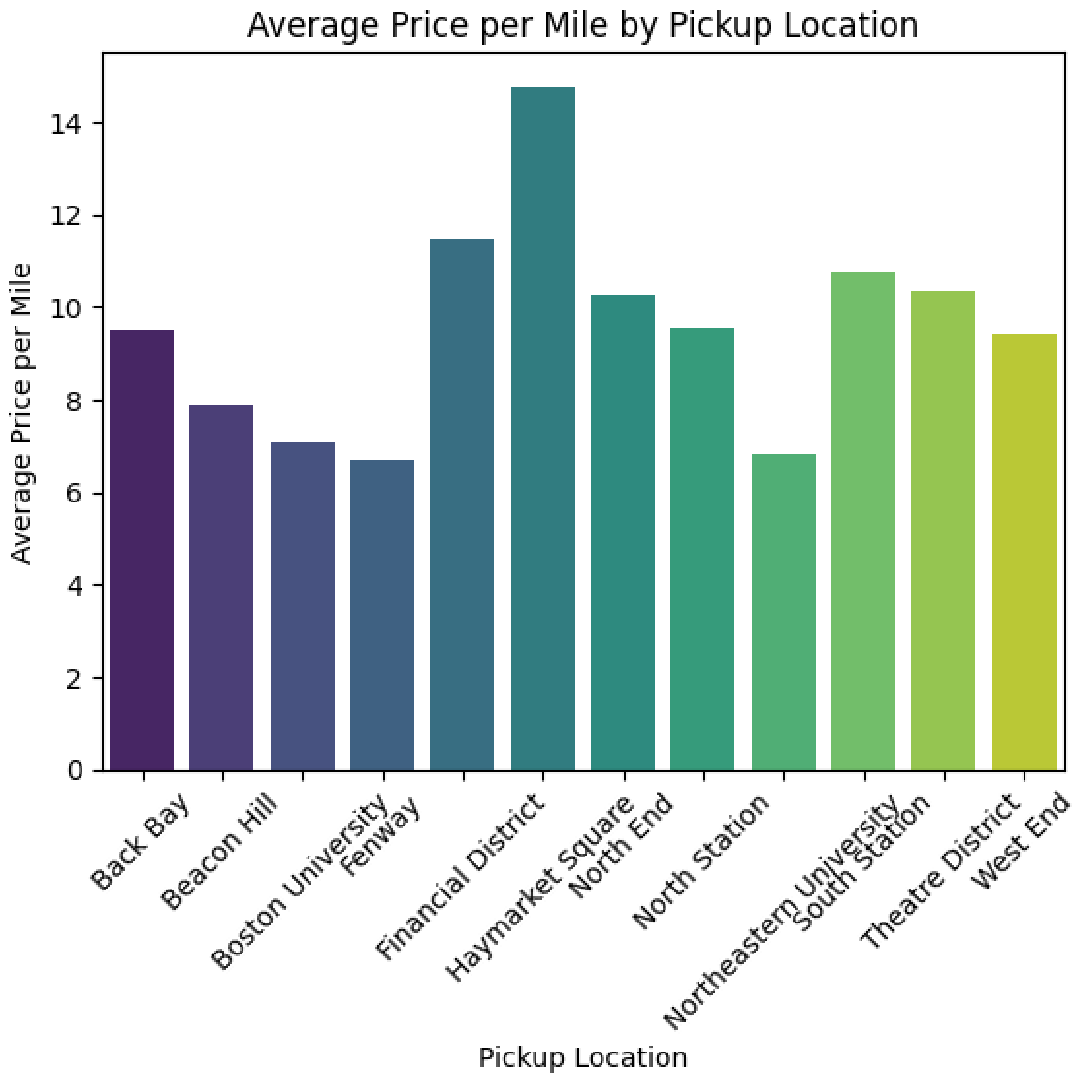
```
<folium.folium.Map at 0x2b216f4ad60>
```

This is a map showing the pickup and dropoff locations on a Map of Boston City, along with it's land boundaries.

Location vs Avg. Price Analysis

```
# Calculate price per mile/km if you have a 'distance' column
df['price_per_mile'] = df['base_price'] / df['distance']

# Group by pickup location
avg_price_per_mile = df.groupby('source')
['price_per_mile'].mean().reset_index()

# Plot the average price per mile by location
sns.barplot(x='source', y='price_per_mile', data=avg_price_per_mile,
palette="viridis")
plt.xticks(rotation=45)
plt.xlabel('Pickup Location')
plt.ylabel('Average Price per Mile')
plt.title('Average Price per Mile by Pickup Location')
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\3212894949.py:8: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(x='source', y='price_per_mile', data=avg_price_per_mile,
palette="viridis")
```



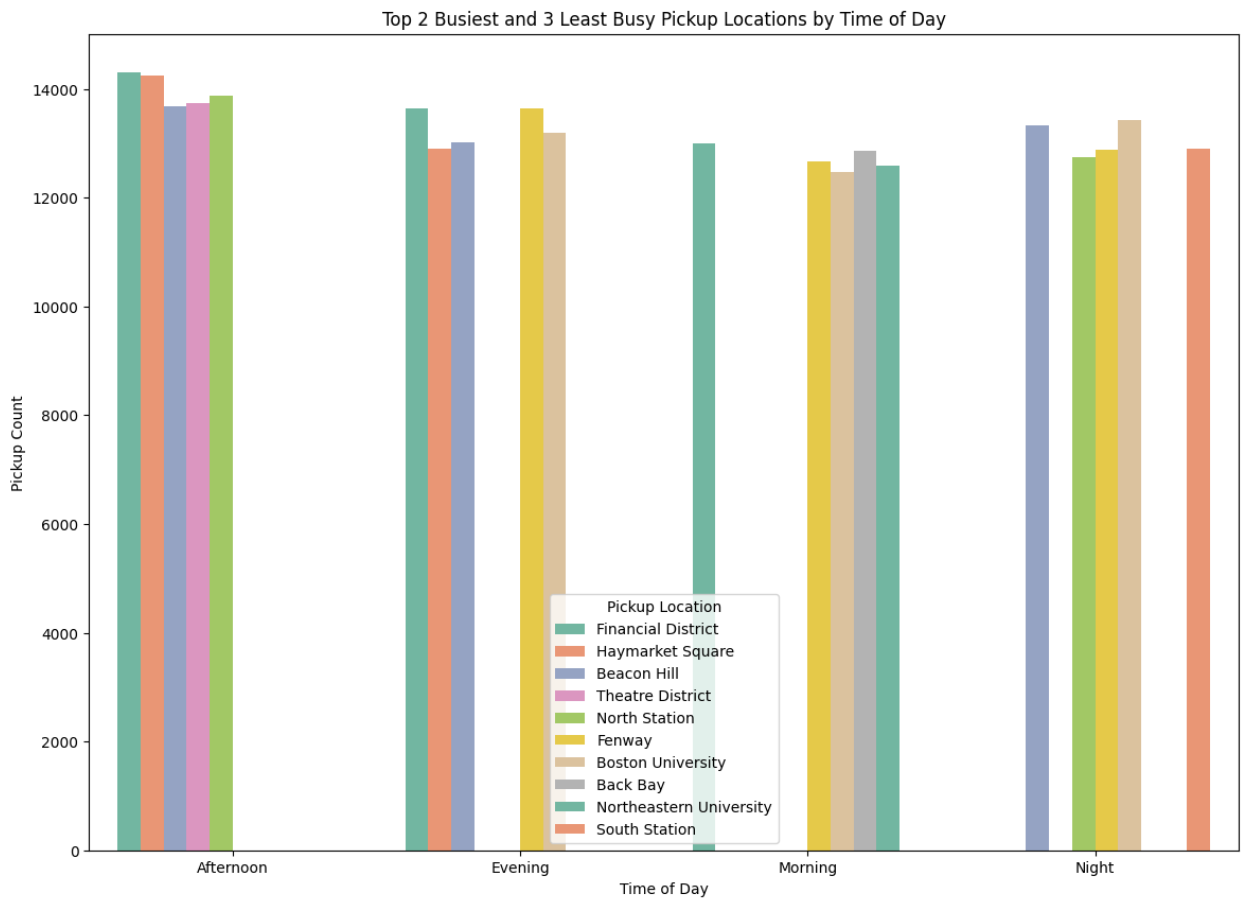
This graph shows that the most expensive pickup location was Financial District, whereas the cheapest were Fenway Park and BU.

Busiest and least busiest pickup spots by time of the day

```
# Step 1: Define the time categories
def categorize_time_of_day(hour):
    if 6 <= hour < 12:
        return 'Morning'
    elif 12 <= hour < 18:
        return 'Afternoon'
    elif 18 <= hour < 24:
        return 'Evening'
```

```
    else:  # This is for hours between 0 and 6 (Night)
        return 'Night'

# Step 2: Apply the function to create a new column 'time_of_day'
df['time_of_day'] = df['hour'].apply(categorize_time_of_day)

# Step 3: Group by time_of_day and pickup_location, then count pickups
hourly_pickups = df.groupby(['time_of_day',
'source']).size().reset_index(name='pickup_count')

# Step 4: Sort values to get the busiest locations by time_of_day
hourly_pickups_sorted = hourly_pickups.sort_values(['time_of_day',
'pickup_count'], ascending=[True, False])

# Step 5: Get the top 2 busiest and 3 least busy locations for each
time of day
top_and_least_pickups =
hourly_pickups_sorted.groupby('time_of_day').apply(
    lambda x: pd.concat([x.nlargest(2, 'pickup_count'), x.nsmallest(3,
'pickup_count')])
).reset_index(drop=True)

# Visualization
plt.figure(figsize=(14, 10))
sns.barplot(
    data=top_and_least_pickups,
    x='time_of_day',
    y='pickup_count',
    hue='source',
    palette='Set2'
)
plt.title("Top 2 Busiest and 3 Least Busy Pickup Locations by Time of
Day")
plt.xlabel("Time of Day")
plt.ylabel("Pickup Count")
plt.legend(title='Pickup Location')
plt.show()
```

```
C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\3410434417.py:22: DeprecationWarning:
DataFrameGroupBy.apply operated on the grouping columns. This behavior
is deprecated, and in a future version of pandas the grouping columns
will be excluded from the operation. Either pass
`include_groups=False` to exclude the groupings or explicitly select
the grouping columns after groupby to silence this warning.
  top_and_least_pickups =
hourly_pickups_sorted.groupby('time_of_day').apply(
```

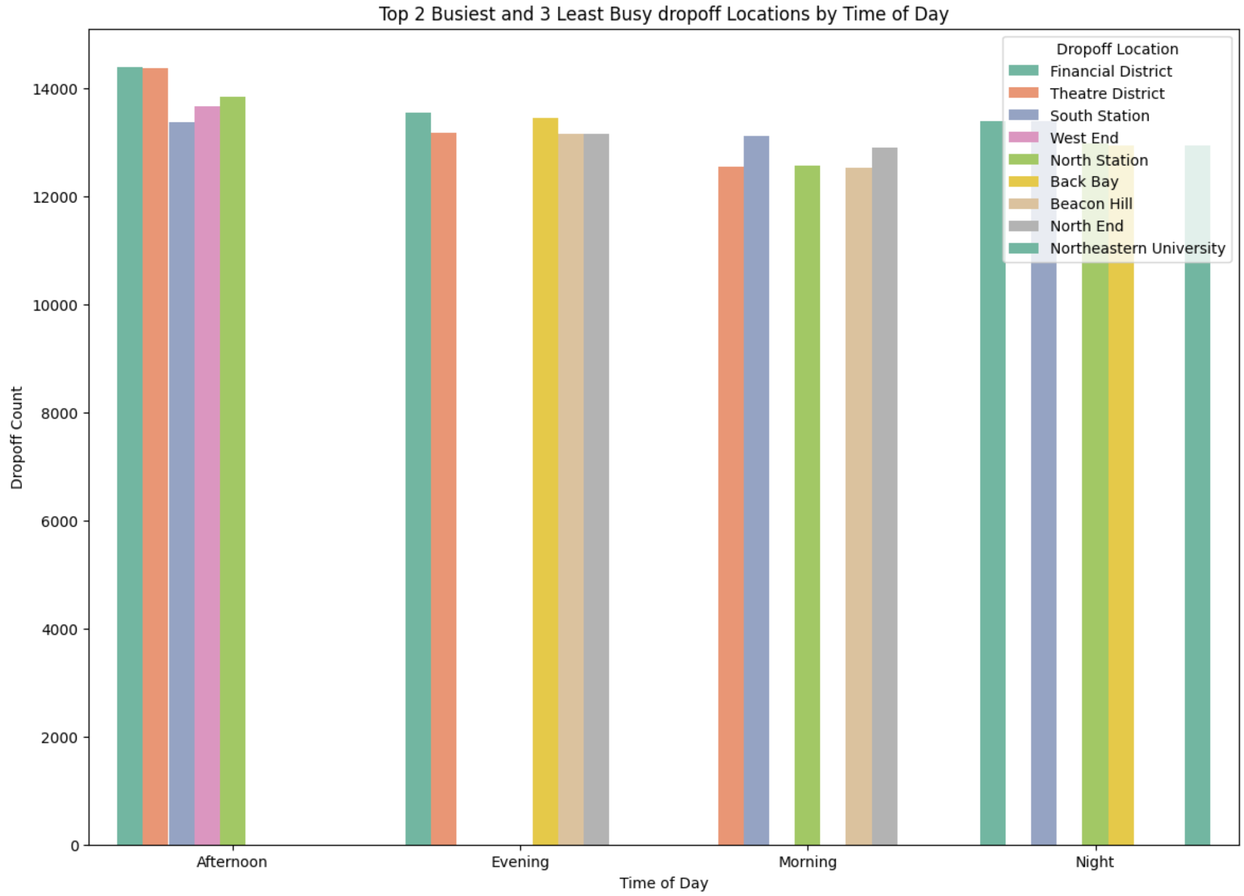


This visualization doesn't give any interesting results

Busiest and least busiest dropoff spots according to time of the Day

```
# Step 1: Define the time categories
def categorize_time_of_day(hour):
    if 6 <= hour < 12:
        return 'Morning'
    elif 12 <= hour < 18:
        return 'Afternoon'
    elif 18 <= hour < 24:
        return 'Evening'
    else:  # This is for hours between 0 and 6 (Night)
        return 'Night'

# Step 2: Apply the function to create a new column 'time_of_day'
df['time_of_day'] = df['hour'].apply(categorize_time_of_day)

# Step 3: Group by time_of_day and pickup_location, then count pickups
```

```
hourly_pickups = df.groupby(['time_of_day',
'destination']).size().reset_index(name='dropoff_count')

# Step 4: Sort values to get the busiest locations by time_of_day
hourly_pickups_sorted = hourly_pickups.sort_values(['time_of_day',
'dropoff_count'], ascending=[True, False])

# Step 5: Get the top 2 busiest and 3 least busy locations for each
time of day
top_and_least_pickups =
hourly_pickups_sorted.groupby('time_of_day').apply(
    lambda x: pd.concat([x.nlargest(2, 'dropoff_count'),
x.nsmallest(3, 'dropoff_count')])
).reset_index(drop=True)

# Visualization
plt.figure(figsize=(14, 10))
sns.barplot(
    data=top_and_least_pickups,
    x='time_of_day',
    y='dropoff_count',
    hue='destination',
    palette='Set2'
)
plt.title("Top 2 Busiest and 3 Least Busy dropoff Locations by Time of
Day")
plt.xlabel("Time of Day")
plt.ylabel("Dropoff Count")
plt.legend(title='Dropoff Location')
plt.show()

C:\Users\Kinshuk Mangal\AppData\Local\Temp\
ipykernel_39840\1810654456.py:22: DeprecationWarning:
DataFrameGroupBy.apply operated on the grouping columns. This behavior
is deprecated, and in a future version of pandas the grouping columns
will be excluded from the operation. Either pass
`include_groups=False` to exclude the groupings or explicitly select
the grouping columns after groupby to silence this warning.
  top_and_least_pickups =
hourly_pickups_sorted.groupby('time_of_day').apply(
```



This visualization doesn't give any interesting results in particular either.

```
import branca.colormap as cm

# Step 1: Calculate Price per Mile
# Assuming `df` contains 'source', 'price', and 'distance' columns
avg_price_per_mile = df.groupby('source')
['price_per_mile'].mean().reset_index()
avg_price_per_mile.columns = ['source', 'avg_price_per_mile']

# Step 2: Prepare GeoData with Latitude and Longitude
# Assuming `coords_df` is a dictionary with coordinates of each
location
coords_df1 = pd.DataFrame.from_dict({
    0: {'location': 'Haymarket Square', 'latitude': 42.361764,
'longitude': -71.057551},
    1: {'location': 'Back Bay', 'latitude': 42.350707, 'longitude': -
71.079730},
    2: {'location': 'North End', 'latitude': 42.365097, 'longitude': -
71.054495},
    3: {'location': 'North Station', 'latitude': 42.366300,
'longitude': -71.062220},
    4: {'location': 'Beacon Hill', 'latitude': 42.358300, 'longitude':
```

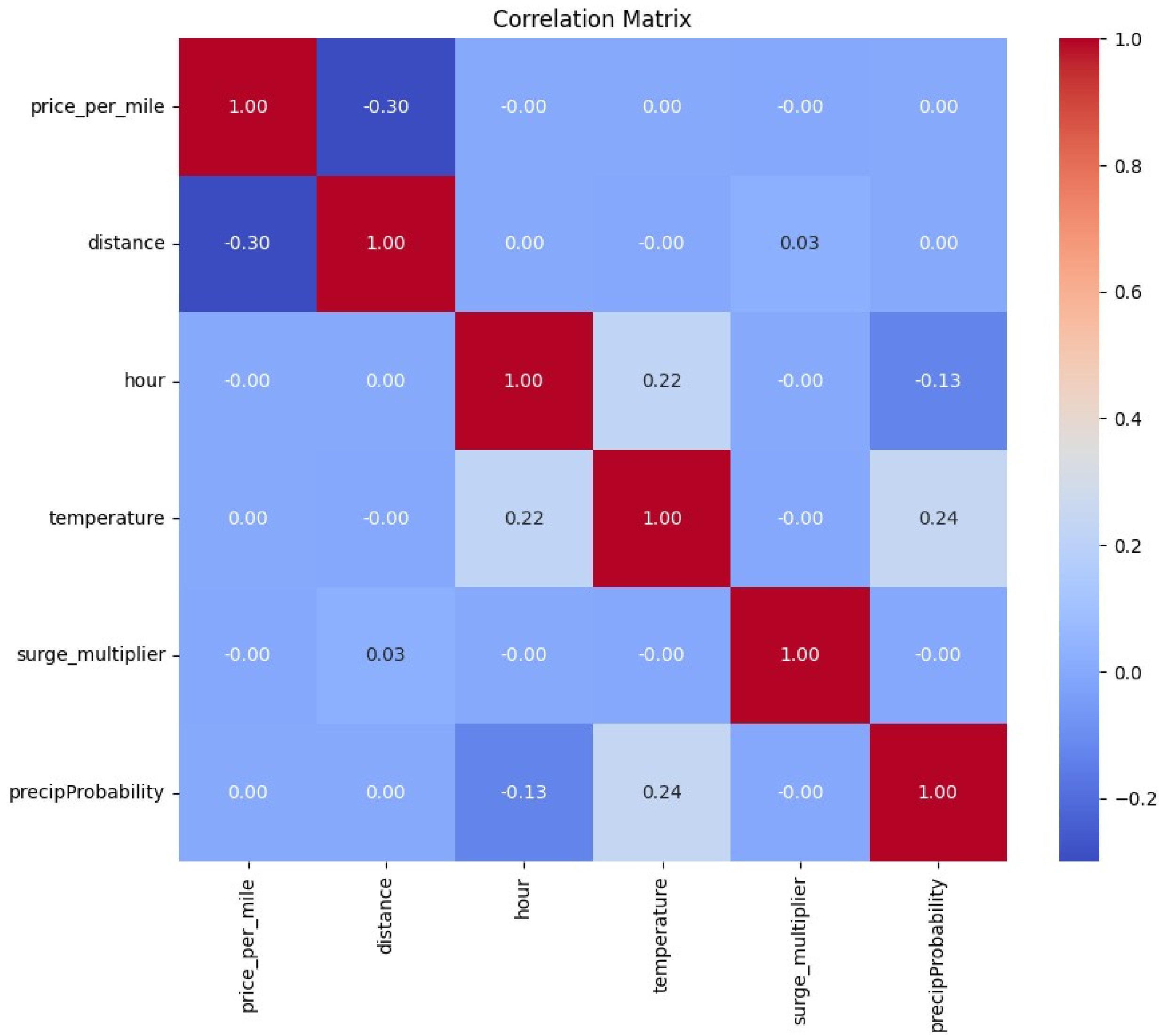
```
-71.066100},
    5: {'location': 'Boston University', 'latitude': 42.350422,
'longitude': -71.103228},
    6: {'location': 'Fenway', 'latitude': 42.345187, 'longitude': -
71.104599},
    7: {'location': 'South Station', 'latitude': 42.352508,
'longitude': -71.054945},
    8: {'location': 'Theatre District', 'latitude': 42.353890,
'longitude': -71.062780},
    9: {'location': 'West End', 'latitude': 42.364758, 'longitude': -
71.067421},
    10: {'location': 'Financial District', 'latitude': 42.355840,
'longitude': -71.055620},
    11: {'location': 'Northeastern University', 'latitude': 42.338954,
'longitude': -71.088058}
}, orient='index')

# Rename columns for merging and prepare geo_data
coords_df1.columns = ['location', 'latitude', 'longitude']
geo_data = pd.merge(avg_price_per_mile, coords_df1, left_on='source',
right_on='location', how='left')

# Step 3: Create a Folium Map
# Initialize the map centered around Boston
m = folium.Map(location=[42.3601, -71.0589], zoom_start=12)

# Set up color map based on average price per mile
colormap = cm.LinearColormap(['green', 'yellow', 'red'],
vmin=geo_data['avg_price_per_mile'].min(),
vmax=geo_data['avg_price_per_mile'].max())
colormap.caption = 'Average Price per Mile'
m.add_child(colormap)

# Add each pickup location to the map
for idx, row in geo_data.iterrows():
    folium.CircleMarker(
        location=[row['latitude'], row['longitude']],
        radius=8,
        color=colormap(row['avg_price_per_mile']),
        fill=True,
        fill_color=colormap(row['avg_price_per_mile']),
        fill_opacity=0.7,
        popup=f"{row['location']}: ${row['avg_price_per_mile']:.2f}
per mile"
    ).add_to(m)

# Display the map
m
```

```
<folium.folium.Map at 0x2b20cbf2490>
```

This visualization shows a chloropleth for the price per mile for each pick-up location.

Correlation Matrix

```
# Step 1: Select numerical columns, including 'price_per_mile'
# Adjust the list
numerical_df = df[['price_per_mile', 'distance', 'hour',
'temperature','surge_multiplier','precipProbability']]

# Step 2: Calculate the correlation matrix
correlation_matrix = numerical_df.corr()

# Step 3: Visualize the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
fmt='.2f')
plt.title("Correlation Matrix")
plt.show()
```



This correlation matrix shows that there are no interesting correlations in the Dataset that we have.

ML (Estimating price_per_mile) [Model 1-General Model]

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# Define features and target variable
```

```
X = df[['source', 'distance',
'hour','cab_type','name','surge_multiplier','temperature']]
y = df['price_per_mile']

# Encode categorical variables and scale numerical features
preprocessor = ColumnTransformer([
    ('num', StandardScaler(),
['distance','hour','surge_multiplier','temperature']),
    ('cat', OneHotEncoder(drop='first'), ['source','cab_type','name'])
])

# Step 2: Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Step 3: Train the Linear Regression model in a pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

pipeline.fit(X_train, y_train)

# Step 4: Evaluate the Model
y_pred = pipeline.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")

#print(max(df['price_per_mile']))
#print(min(df['price_per_mile']))
```

```
Mean Absolute Error (MAE): 2.91
Mean Squared Error (MSE): 108.57
```

Here we are using ML using the scikit-learn library that splits the data into test and training sets. We use both numerical and categorical variables to estimate the price per mile with an MAE of $2.91.

Predicting the Final Price based on Model 1

```
import numpy as np

def predict_final_price():
    # Ask for user input
    source = input("Enter the source (e.g., A, B, C): ")
    distance = float(input("Enter the distance in miles: "))
```

```
    hour = int(input("Enter the hour of the day (0-23): "))
    cab_type = input("Enter the cab company (e.g., Uber, Lyft): ")
    name = input("Enter the cab service (e.g., Uber, UberXL, etc.): ")
    surge_multiplier = float(input("Enter the surge multiplier (e.g., 
1.2): "))
    temperature = float(input("Enter the temperature (in Fahrenheit): 
"))

    # Create a DataFrame with the user input data
    input_data = pd.DataFrame([[source, distance, hour, cab_type, 
name, surge_multiplier, temperature]],
                              columns=['source', 'distance', 'hour', 
'cab_type', 'name', 'surge_multiplier', 'temperature'])

    # Use the pipeline to predict the price per mile for the user 
input
    predicted_price_per_mile = pipeline.predict(input_data)[0]

    # Calculate the final price (price per mile * distance)
    final_price = predicted_price_per_mile * distance

    print(f"The predicted final price is: ${final_price:.2f}")

# Call the function to run the prediction
predict_final_price()

Enter the source (e.g., A, B, C):  Haymarket Square
Enter the distance in miles:  5
Enter the hour of the day (0-23):  22
Enter the cab company (e.g., Uber, Lyft):  Lyft
Enter the cab service (e.g., Uber, UberXL, etc.):  Lyft
Enter the surge multiplier (e.g., 1.2):  1.0
Enter the temperature (in Fahrenheit):  35

The predicted final price is: $-20.78
```

This function allows us to calculate the estimates price for the total ride, by inputting the desired factors.

Filtering outliers for price_per_mile

```
# Step 1: Calculate Q1 (25th percentile) and Q3 (75th percentile)
Q1 = df['price_per_mile'].quantile(0.25)
Q3 = df['price_per_mile'].quantile(0.75)

# Step 2: Calculate the Interquartile Range (IQR)
IQR = Q3 - Q1
```

```
# Step 3: Define the lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Step 4: Create a new column 'is_outlier' where True means the value
is an outlier
df['is_outlier'] = (df['price_per_mile'] < lower_bound) |
(df['price_per_mile'] > upper_bound)
```

ML Model 2- [Working with non-outlier Data only]

```
# Step 1: Filter out the outliers from the dataset based on
price_per_mile
non_outlier_data = df[df['is_outlier'] == False]

# Step 2: Define the features (X) and target (y) for the non-outlier
dataset
X_non_outlier = non_outlier_data.drop(columns=['price_per_mile',
'is_outlier'])  # Keep all features except target and outlier
indicator
y_non_outlier = non_outlier_data['price_per_mile']  # Target is
price_per_mile

# Step 3: Split the data into training and test sets
X_train_non_outlier, X_test_non_outlier, y_train_non_outlier,
y_test_non_outlier = train_test_split(
    X_non_outlier, y_non_outlier, test_size=0.2, random_state=42
)

# Step 4: Create the pipeline again (same preprocessor and regressor
as before)
pipeline_non_outlier = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

# Step 5: Train the model on the non-outlier data
pipeline_non_outlier.fit(X_train_non_outlier, y_train_non_outlier)

# Step 6: Predict and evaluate the model on the non-outlier test set
y_pred_non_outlier = pipeline_non_outlier.predict(X_test_non_outlier)

# Step 7: Calculate MAE and MSE for the non-outlier model
mae_non_outlier = mean_absolute_error(y_test_non_outlier,
y_pred_non_outlier)
mse_non_outlier = mean_squared_error(y_test_non_outlier,
```

```
y_pred_non_outlier)

print(f"Non-Outlier Model - Mean Absolute Error (MAE): 
{mae_non_outlier:.2f}")
print(f"Non-Outlier Model - Mean Squared Error (MSE): 
{mse_non_outlier:.2f}")

Non-Outlier Model - Mean Absolute Error (MAE): 1.31
Non-Outlier Model - Mean Squared Error (MSE): 3.26
```

This model works only with the non-outlier data. Here we are able to lower the MAE of price-per-mile to $1.31. Which in my opinion is pretty good for a very rudimentary ML model.

Predicting the final price based on Model 2

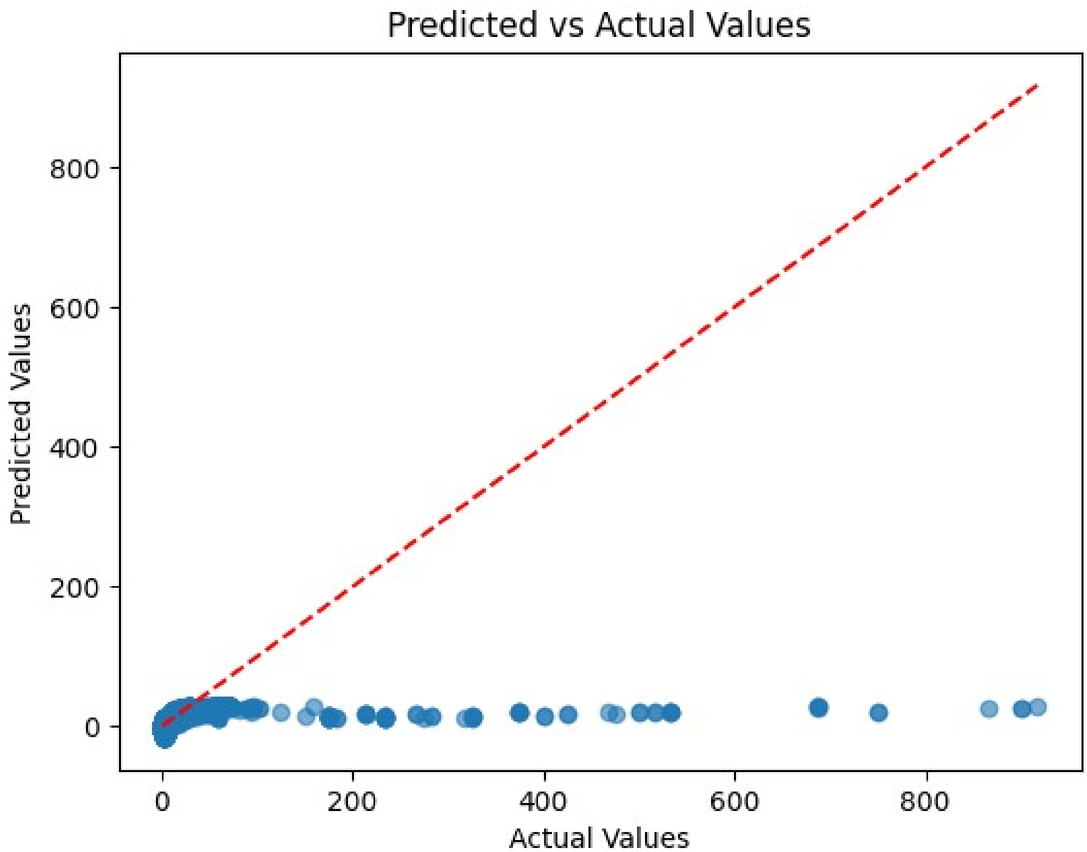
```
def predict_final_price_non_outlier():
    # Ask for user input
    source = input("Enter the source (e.g., A, B, C): ")
    distance = float(input("Enter the distance in miles: "))
    hour = int(input("Enter the hour of the day (0-23): "))
    cab_type = input("Enter the cab company (e.g., Uber, Lyft): ")
    name = input("Enter the cab service (e.g., Uber, UberXL, etc.): ")
    surge_multiplier = float(input("Enter the surge multiplier (e.g., 
1.2): "))
    temperature = float(input("Enter the temperature (in Fahrenheit): 
"))

    # Create a DataFrame with the user input data
    input_data = pd.DataFrame([[source, distance, hour, cab_type, 
name, surge_multiplier, temperature]],
                              columns=['source', 'distance', 'hour', 
'cab_type', 'name', 'surge_multiplier', 'temperature'])

    # Use the trained pipeline to predict the price per mile for the 
user input
    predicted_price_per_mile = 
pipeline_non_outlier.predict(input_data)[0]

    # Calculate the final price (price per mile * distance)
    final_price = predicted_price_per_mile * distance

    print(f"The predicted final price is: ${final_price:.2f}")

# Call the function to run the prediction
predict_final_price_non_outlier()

Enter the source (e.g., A, B, C):  Haymarket Square
Enter the distance in miles:  8
```

```
Enter the hour of the day (0-23):  11
Enter the cab company (e.g., Uber, Lyft):  Uber
Enter the cab service (e.g., Uber, UberXL, etc.):  UberXL
Enter the surge multiplier (e.g., 1.2):  1.0
Enter the temperature (in Fahrenheit):  40

The predicted final price is: $-19.98
```

This function helps in predicting the price for the total journey for the 2nd model.

Model 1 Predicted vs Actual Scatterplot

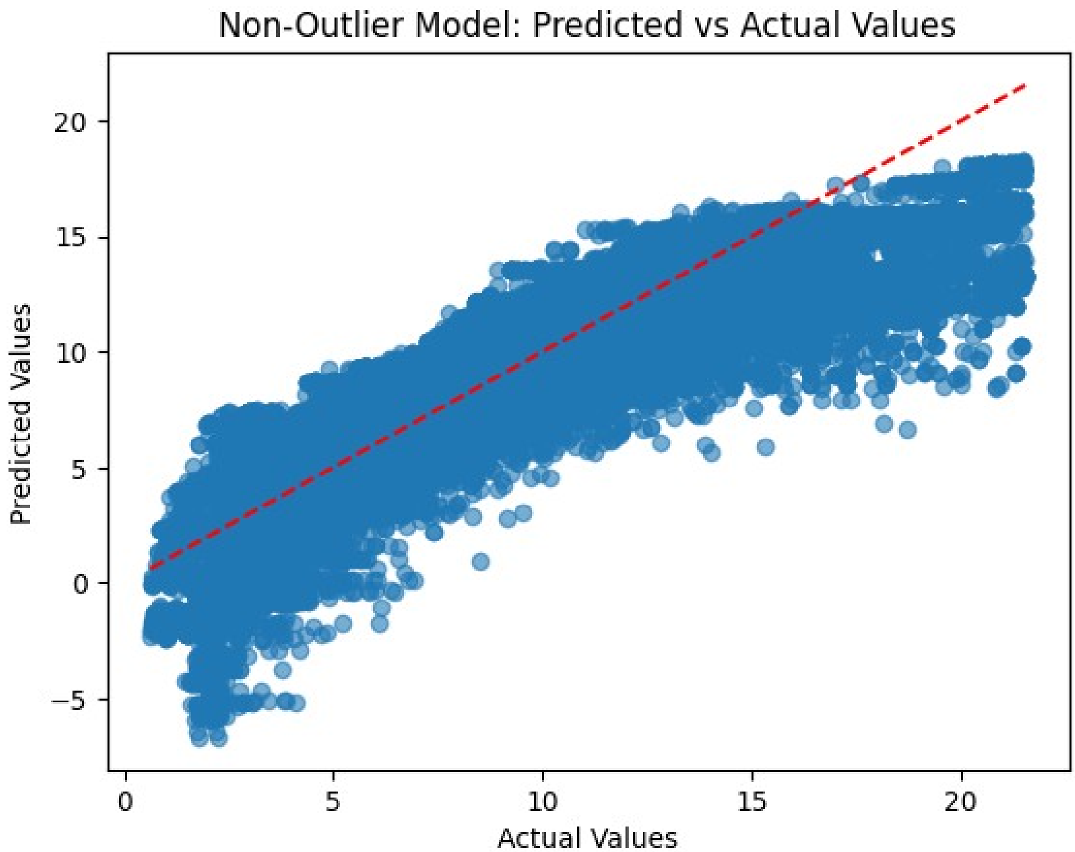
```python
import matplotlib.pyplot as plt

# Scatter plot for predicted vs actual values
plt.scatter(y_test, y_pred, alpha=0.6)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)],
color='red', linestyle='--')  # Line of perfect prediction
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Predicted vs Actual Values')
plt.show()
```



This plot shows that for less extreme values, this model predicts decently. However, for extreme points, it starts to perform poorly.

Model 2 Predicted vs Actual scatterplot

```
# Scatter plot for predicted vs actual values
plt.scatter(y_test_non_outlier, y_pred_non_outlier, alpha=0.6)
plt.plot([min(y_test_non_outlier), max(y_test_non_outlier)],
[min(y_test_non_outlier), max(y_test_non_outlier)], color='red',
linestyle='--')  # Line of perfect prediction
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Non-Outlier Model: Predicted vs Actual Values')
plt.show()
```

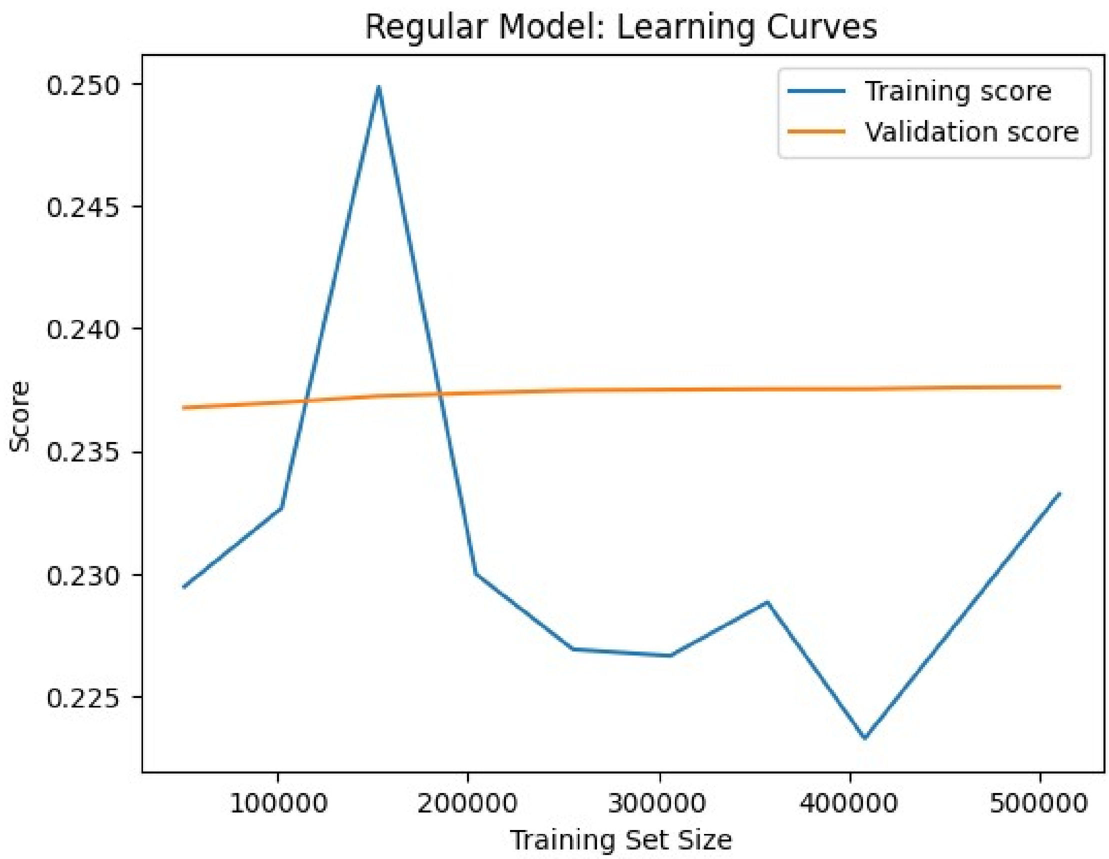


The model 2 performs a lot better with the outliers gone.

Learning Curve for Model 1

```
# Define features (X) and target (y) for the regular dataset
(including both outlier and non-outlier data)
X_regular = df.drop(columns=['price_per_mile', 'is_outlier'])  # Keep
all features except target and outlier indicator
y_regular = df['price_per_mile']  # Target is price_per_mile

# Learning curve
train_sizes, train_scores, validation_scores = learning_curve(
    pipeline, X_regular, y_regular, cv=5, n_jobs=-1,
    train_sizes=np.linspace(0.1, 1.0, 10))  # Train sizes range from
10% to 100%

# Plot the learning curve
plt.plot(train_sizes, np.mean(train_scores, axis=1), label="Training
score")
plt.plot(train_sizes, np.mean(validation_scores, axis=1),
label="Validation score")
plt.xlabel('Training Set Size')
```

```
plt.ylabel('Score')
plt.title('Regular Model: Learning Curves')
plt.legend()
plt.show()
```

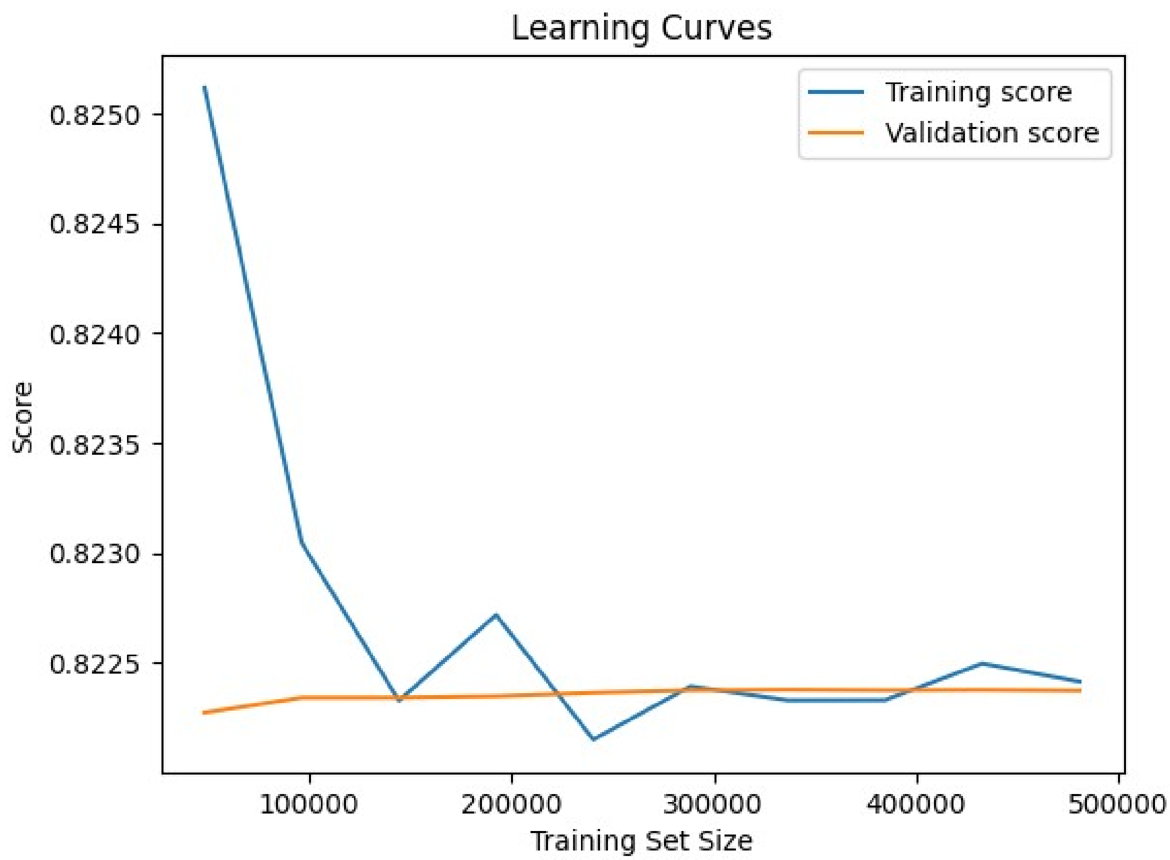


This is just another way of visualizing the performance of Model 1, we can see that there's a large difference between the training score and validation score.

Learning curve for the Model 2

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, validation_scores = learning_curve(
    pipeline_non_outlier, X_non_outlier, y_non_outlier, cv=5, n_jobs=-
1,
    train_sizes=np.linspace(0.1, 1.0, 10))

# Plot the learning curve
plt.plot(train_sizes, np.mean(train_scores, axis=1), label="Training
score")
plt.plot(train_sizes, np.mean(validation_scores, axis=1),
```

```
label="Validation score")
plt.xlabel('Training Set Size')
plt.ylabel('Score')
plt.title('Learning Curves')
plt.legend()
plt.show()
```



The 2nd model in general is a lot more coinciding with the actual values, and thus performs better.

```

```