Ryde Automation EDA and Business Actions

Andrew Wilson

10/31/2018

Prompt

As the newest data analytics hire at an autonomous vehicle startup (Ryde Automation), you have been given an example dataset (~1000 rows, 8 columns) and are asked to suggest some business actions for Ryde based on your conclusions. We encourage you to explore the data and move in any direction that you think will benefit the company the most. If you are running short on time, please document your thought process in words and explain in detail how you would proceed if given additional time.

There is a small intentional error in the dataset to look out for! (Let us know in the report if you find it and fix it!)

This take-home exercise is meant to be open-ended. Please feel free to use whatever tools you are most comfortable with and please submit your final report in any format that best conveys your findings.

Disclaimer: This dataset has no affiliation with the data at Cruise Automation.

Table schema description:

- *user_id: identification number assigned to each user*
- *car_id: name of each car used for the ride*
- *start_time: start time of the ride in YYYY-MM-DD HH:MM:SS*
- *end_time: end time of the ride in YYYY-MM-DD HH:MM:SS*
- *num_riders: number of passengers in the ride*
- *region: the region that the car is operating in (origin)*
- *num_near_misses: the number of times that the car is close to getting into an accident during the ride*
- *price: the price that the rider paid to get on the ride*
- *rating: the rating (1 through 5) that the rider gave the ride (5 is best)*

Executive summary

This report takes a hypothetical autonomous vehicle startup data set, analyzes it, and offers business actions based on the results. The report is split into five major sections, each containing some background information, code (process/methodology), results, and recommendations:

- Setup, load, and clean — get dataset into a workable state
- Safety — reduce the number of near misses
- Ratings — causes of bad ratings and how to improve them
- Utilization — dive deep into quantifying the amount of empty car seats
- Pricing — current pricing structure and how to optimize it

Setup, load, and clean

Setup

The first step is to set up the environment properly:

- Remove all environment variables so that program always has the same end result given the same input data
- Set up default options for publishing from R markdown to PDF
- Load all necessary libraries, including ability to code Python in R Markdown
- Set up a custom theme for all ggplot charts
- Change some of the default settings

```
## clean workspace
rm(list=ls(all=TRUE))

# knitr setup
knitr::opts_chunk$set(
    echo = TRUE,
    message = FALSE,
    warning = FALSE,
    error = TRUE,
    comment = "#>",
    tidy.opts = list(width.cutoff = 80)
)
 # keep output with code (or not if FALSE)
knitr::opts_chunk$set(tidy.opts=list(width.cutoff=80))

# load libraries
library(scales)
library(GGally)
library(RColorBrewer)
library(readxl)
library(knitr)
library(scales)
library(reticulate)
use_python("/Users/andrewwilson/anaconda/bin/python", required=TRUE)
#library(officer)
library(lubridate)
library(tidyverse) # make sure piping function loaded last
#library(magrittr) # always have piping available


## set custom colors
custom_colors <- c("#fc553c", "#553d65")

## set custom theme
custom_theme <- function(base_size = 14, base_family = "Helvetica", ...){
  modifyList(theme_minimal(base_size = base_size, base_family = base_family),
             list(
               legend.background =  element_rect(color = "black")
             )
  )
}
```

```r
theme_set(custom_theme())

## disable scientific notation
options(scipen=999)
```

Data import

Importing data is relatively straightforward. Simply set up the appropriate directories and read in the data. It's important to look at what fields/features are available before beginning to clean the data.

```r
## set up data directories, files names, and paths
#root_dir <- "/Users/andrewwilson/Documents-backup/Projects/ryde-take-home/"
data_file_name <- "final_analytics_takehome.xlsx"
data_path <- str_c("data/", data_file_name)

## import data from Excel
df_raw <- read_excel(data_path)

# check out what data we're working with
glimpse(df_raw)
```

```
#> Observations: 1,033
#> Variables: 9
#> $ user_id         <dbl> 9, 12, 3, 10, 9, 15, 15, 19, 14, 8, 16, 20, 11...
#> $ car_id          <chr> "spiderman", "superman", "hulk", "spiderman", ...
#> $ start_time      <dttm> 2018-10-02 03:00:21, 2018-10-02 03:01:30, 201...
#> $ end_time        <dttm> 2018-10-02 03:08:19, 2018-10-02 03:09:16, 201...
#> $ num_riders      <dbl> 3, 2, 2, 3, 4, 3, 5, 1, 3, 4, 3, 2, 1, 2, 5, 5...
#> $ region          <chr> "sf", "sf", "sf", "sf", "sf", "south sf", "sf"...
#> $ num_near_misses <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
#> $ price           <dbl> 3.25, 2.95, 3.22, 2.29, 2.93, 2.48, 3.42, 5.30...
#> $ rating          <dbl> 5, 5, 5, 5, 1, 5, 5, 5, 3, 5, 3, 5, 5, 3, 5, 5...
```

Data cleaning

Cleaning data happens iteratively during the exploration process.

Data exploration in this phase exposed some interesting characteristics of the data set. Here are some general findings about the shape of the data:

- All rides are from October 2, 2018 at 3:00:21 to 23:59:21 on the same day.
- There is no pricing between $3.50 and $5.00
- There are no ride times between 30 and 60 minutes

Features can be added that are clear transformations from the existing data. I added a duration feature called `ride_mins`, and two categorical features to label the time and price as "short"/"long" and "cheap"/"expensive", respectively. Others that aren't necessary can be removed.

Anomalies and/or outliers should first be checked to make sure they aren't in fact true. If they are, there could be some interesting learnings there. If it's an obvious error, this data can be removed from the analysis. Some of the ride_mins were actually negative, so I removed this since it's physically impossible.

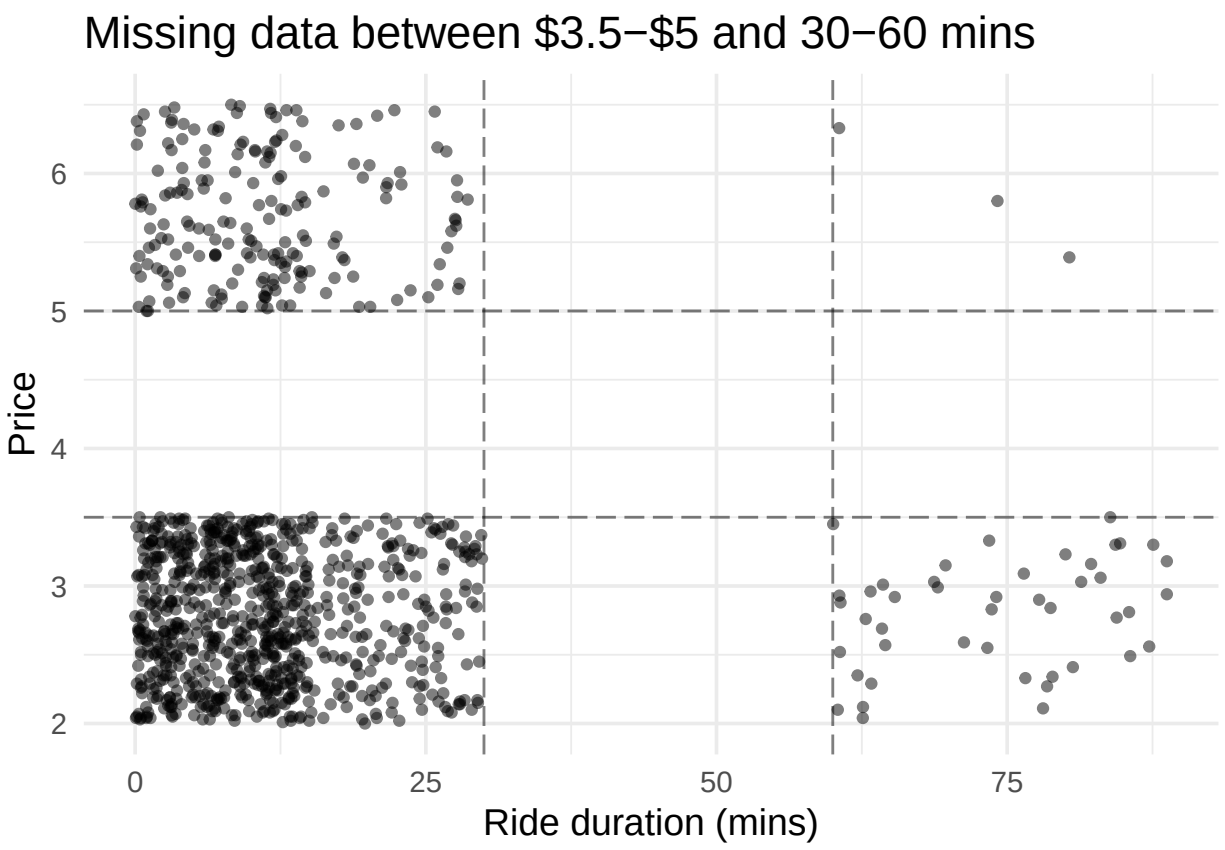
Factors are useful data types in R, especially when it comes to visualizing bar charts. Some of the categorical variables were converted into factors.

It's good to take another peek at the data set once it's been cleaned properly.

```
## initial data cleaning
df <- df_raw %>%

  # change characters to factors
  mutate_if(is.character, as.factor) %>%
  mutate(user_id = as.factor(user_id)) %>%
  mutate(num_riders = as.factor(num_riders)) %>%
  mutate(rating = as.factor(rating)) %>%

  # calculate ride time in minutes
  mutate(ride_mins = as.numeric(end_time - start_time)/60) %>%

  # eliminate data points that have negative ride time
  filter(ride_mins>0) %>%

  # add categorical labels for long, short, cheap, and expensive rides
  mutate(time_cat = ifelse(ride_mins < 45, "short", "long")) %>%
  mutate(price_cat = ifelse(price < 4, "cheap", "expensive"))

# Check out cleansed data
glimpse(df)
```

```
#> Observations: 1,017
#> Variables: 12
#> $ user_id         <fct> 9, 12, 3, 10, 9, 15, 15, 19, 14, 8, 16, 20, 11...
#> $ car_id          <fct> spiderman, superman, hulk, spiderman, scarecro...
#> $ start_time      <dttm> 2018-10-02 03:00:21, 2018-10-02 03:01:30, 201...
#> $ end_time        <dttm> 2018-10-02 03:08:19, 2018-10-02 03:09:16, 201...
#> $ num_riders      <fct> 3, 2, 2, 3, 4, 3, 5, 1, 3, 4, 3, 2, 1, 2, 5, 5...
#> $ region          <fct> sf, sf, sf, sf, sf, south sf, sf, sf, sf, sout...
#> $ num_near_misses <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
#> $ price           <dbl> 3.25, 2.95, 3.22, 2.29, 2.93, 2.48, 3.42, 5.30...
#> $ rating          <fct> 5, 5, 5, 5, 1, 5, 5, 5, 3, 5, 3, 5, 5, 3, 5, 5...
#> $ ride_mins       <dbl> 7.9666667, 7.7666667, 7.5000000, 13.4333333, 6...
#> $ time_cat        <chr> "short", "short", "short", "short", "long", "s...
#> $ price_cat       <chr> "cheap", "cheap", "cheap", "cheap", "cheap", "...
```

```
# visualize missing price and duration values
ggplot(df, aes(ride_mins, price)) +
  geom_point(alpha=0.5) +
  geom_vline(xintercept=c(30, 60), linetype=5, alpha=0.5) +
  geom_hline(yintercept=c(3.5, 5), linetype=5, alpha=0.5) +
  labs(x="Ride duration (mins)",
     y="Price",
     title="Missing data between $3.5-$5 and 30-60 mins")
```



Safety

A top priority for Ryde Automation is to build self driving cars that are **safe**. We'll start by examining the number of near misses (`num_near_misses`). The goal is to better understand when and how near misses occur in the data, to infer the causes, and to suggest possible actions to reduce the number of near misses in the future.

Some notable findings:

- All near misses happened on "long" rides (> 1 hour)
- All near misses happened in SF (not South SF)
- Near misses are always rated as 1 or 2 stars (creating a correlation)
- Superman has the most near misses

Tables to demonstrate:

```
# filter data frame to get only rides with near misses
df_misses <- df %>%
  select(num_near_misses, ride_mins, region, rating, car_id) %>%
  filter(num_near_misses > 0) %>%
  arrange(desc(num_near_misses)) %>%
  mutate(ride_mins = round(ride_mins))
```

```
kable(df_misses, booktabs = TRUE,
      caption="Near misses are on long rides, in SF, and are rated at 1 or 2 stars")
```

Table 1: Near misses are on long rides, in SF, and are rated at 1 or 2 stars

num_near_misses	ride_mins	region	rating	car_id
5	84	sf	1	superman
3	63	sf	1	superman
2	69	sf	1	superman
2	65	sf	1	superman
2	83	sf	1	superman
2	78	sf	1	superman
2	73	sf	1	superman
1	81	sf	1	robin
1	79	sf	2	superman
1	65	sf	1	joker
1	79	sf	2	spiderman
1	80	sf	2	batman
1	74	sf	1	scarecrow
1	64	sf	2	spiderman
1	76	sf	2	ironman
1	61	sf	1	superman
1	61	sf	1	superman
1	78	sf	1	superman

```
# count number of misses per car
df_misses_cars <- df_misses %>% group_by(car_id) %>%
  summarize(total_near_misses = sum(num_near_misses),
            percentage_near_misses = percent(total_near_misses/sum(df_misses$num_near_misses))) %>%
  arrange(desc(total_near_misses))

kable(df_misses_cars, booktabs = TRUE,
      caption="Superman has three quarters of the near misses")
```

Table 2: Superman has three quarters of the near misses

car_id	total_near_misses	percentage_near_misses
superman	22	75.9%
spiderman	2	6.90%
batman	1	3.45%
ironman	1	3.45%
joker	1	3.45%
robin	1	3.45%
scarecrow	1	3.45%

Business actions:

Ryde Automation should make an effort to better understand *why* near misses occur. Since most of the rides are in SF (not South SF), it makes sense that all near misses occur there. It also makes sense that near

misses always occur on "long" rides (>1hr), but it should be investigated given a larger data set to see if this is always the case. Not only are near misses endangering your passengers, but they are also the cause of 85% of 1 star ratings (13/15). The best way to understand why near misses occur is to look more carefully at the `Superman` car, since it is responsible for 3/4 of all near misses. Given more data about this car, we could dig in to better understand the underlying issues to prevent them from happening in other cars.

Ratings

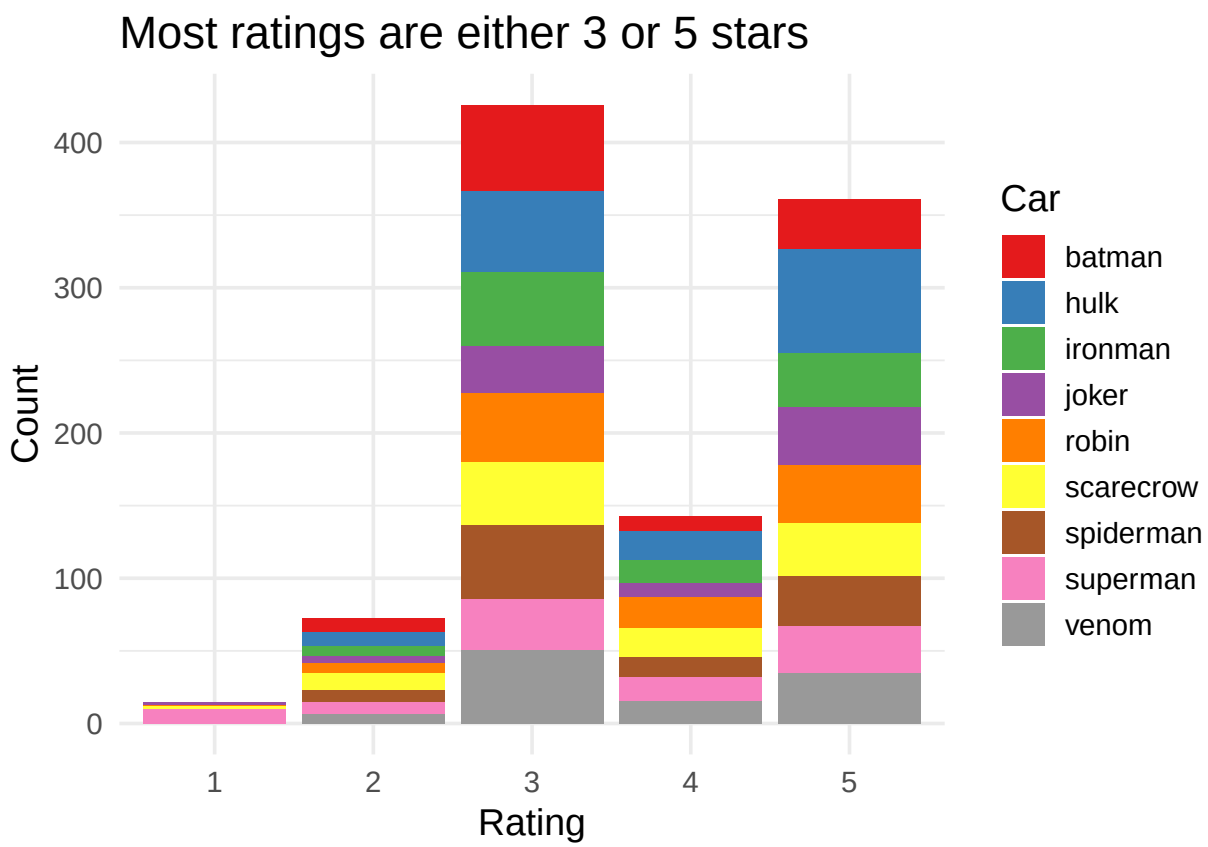
Ratings are a crucial indicator of the quality of service at Ryde Automation. They directly relate to customer retention, a critical business metric for a new ride sharing company. We'll examine this feature next, with the goal of better understanding how ratings are distributed, what causes a good/bad rating, and how to improve ratings in the future.

Some notable findings:

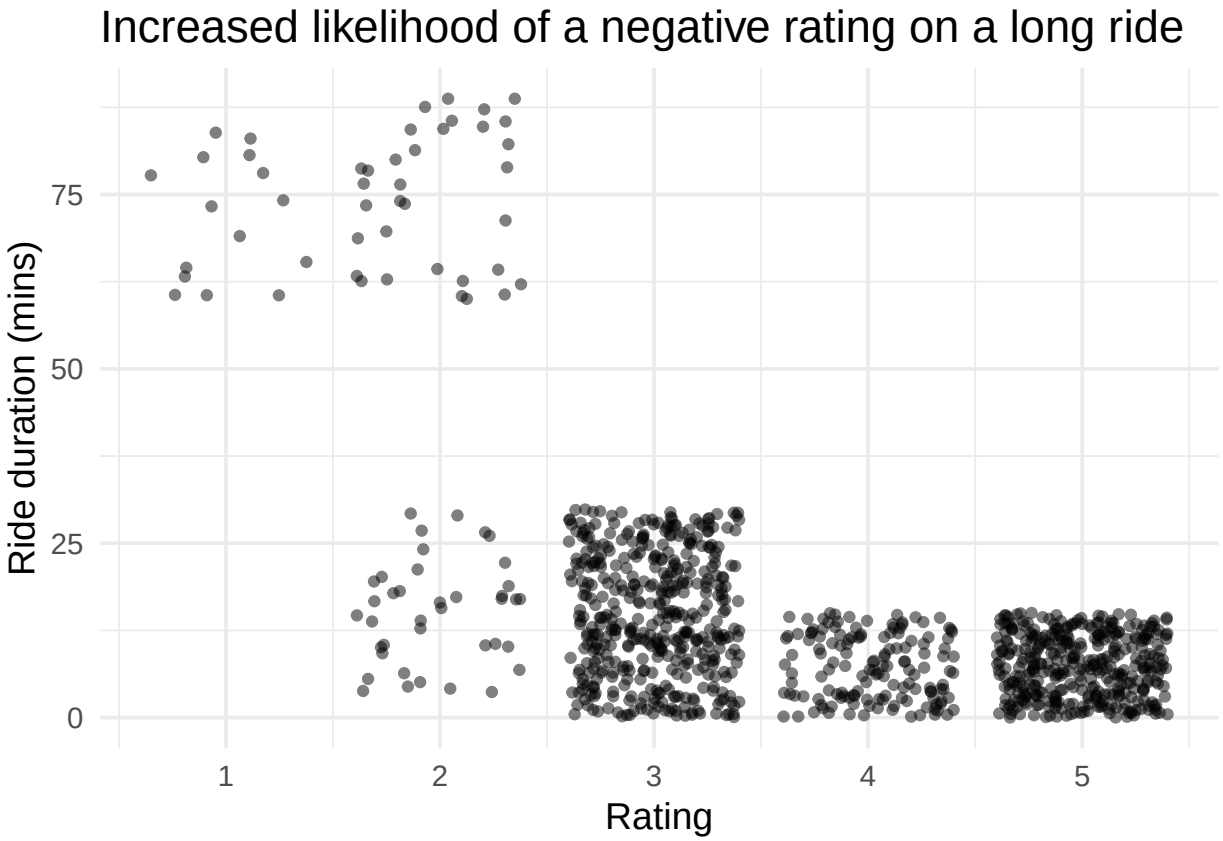
- Most ratings are either 3 or 5 stars
- Superman, as previously discussed, has most of the 1 star ratings (likely due to its numerous near misses)
- Hulk has the most 5 star ratings
- Most negaitve reviews happen on "long" rides (there is a slight negative correlation between `rating` and `ride_mins`)

Charts to demonstrate:

```
ggplot(df, aes(rating, fill = car_id)) +
  geom_bar() +
  scale_fill_brewer(palette="Set1") +
  labs(x="Rating",
       y="Count",
       fill="Car",
       title="Most ratings are either 3 or 5 stars")
```




```
ggplot(df, aes(as.integer(rating), ride_mins)) +
  geom_jitter(alpha = 0.5) +
  labs(x="Rating",
     y="Ride duration (mins)",
     title="Increased likelihood of a negative rating on a long ride")
```



Business actions:

It's clear that Superman is getting 1 star ratings because of its near misses, which should be looked into. All other ratings are distributed relatively evenly between cars (and amongst other factors), but the Hulk is doing particularly well. It might be worth investigating exactly what the Hulk is doing that gets it such glowing ratings. Not much more can be derived from this data set, but it might be worth understanding how to get customers to up their ratings from 3 stars to 4 stars.

Utilization

Introduction and methodology

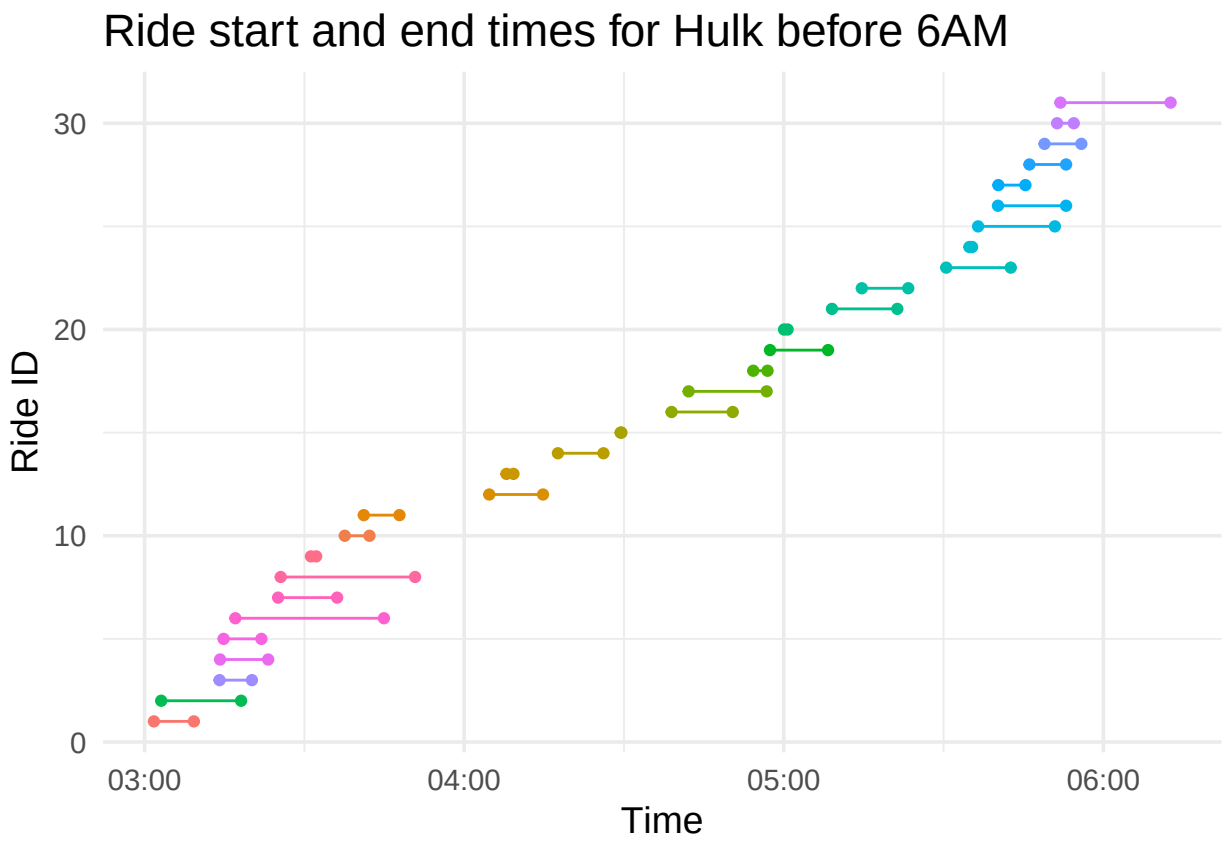
Utilization is a key metric for a ride sharing company, as it determines how much free capacity is available. This enables the company to increase revenues without increasing capital expenditures (extra profit!). In this case, I calculated utilization as the number of seats taken by riders divided by the total number of seats (5 per car) for any given time period.

Unfortunately, calculating utilization is not always as easy as it seems, and it's certainly the case with this data set. There are a few problems that we'll have to deal with in order to get to an accurate utilization number:

- Account for time in-between rides, where there are 0 riders
- Understand how and when rides overlap with one another
- Determine how to calculate utilization as riders get in and out of a car
- Do this analysis for each car, individually and in aggregate

To begin, I created a visualization of individual ride times for a single car within a small time frame. I wanted to better understand how rides overlapped, and how much time there typically was inbetween rides. I also created a table to see the accompanying number of riders (`num_riders`).

```
# filter to one car for a small period of time
df_hulk <- df %>%
  filter(car_id == "hulk") %>%
  filter(start_time < ymd_hms("2018-10-02 6:00:00")) %>%
  mutate(ride_id = row_number()) %>%
  select(ride_id, start_time, end_time, ride_mins, num_riders)

# visualize a few rides for a single car
df_hulk %>%
  gather("startORend", "time", 2:3) %>%
  arrange(ride_id) %>%
  ggplot(aes(time, ride_id, color=as.character(ride_id))) +
  geom_point() +
  geom_line() +
  guides(color=FALSE) +
  labs(x="Time",
     y="Ride ID",
     title="Ride start and end times for Hulk before 6AM")
```




```
# print table of first 5 rides
kable(head(mutate(df_hulk, ride_mins = round(ride_mins))), booktabs = TRUE,
      caption="Rides overlap, and it's not clear how # riders changes")
```

Table 3: Rides overlap, and it's not clear how # riders changes

ride_id	start_time	end_time	ride_mins	num_riders
1	2018-10-02 03:01:45	2018-10-02 03:09:15	8	2
2	2018-10-02 03:03:07	2018-10-02 03:18:07	15	5
3	2018-10-02 03:14:04	2018-10-02 03:20:09	6	5
4	2018-10-02 03:14:10	2018-10-02 03:23:12	9	5
5	2018-10-02 03:14:50	2018-10-02 03:21:56	7	4
6	2018-10-02 03:17:02	2018-10-02 03:44:56	28	1

Algorithm approach

From the chart and the table, we can see that there in fact are overlapping rides, so people do get in and out of the car. However, when we look at the table we see that ride 3 had 5 people, and ride 4 started within that time period, also with 5 people. I'm unsure if this is an error in the data, or if it is recorded correctly, so I will make some assumptions about how to look at the weighted utilization over time given overlapping rides with differing numbers of people.

I decided to create an algorithm, using Python (easier build algorithms and read code), that outputs a flattened version of the rides over time (no overlap). It calculates the average weighted number of people in the ride over a given time period, and also adds in time with "0" riders for the gaps inbetween rides. The code for this algorithm can be seen below:

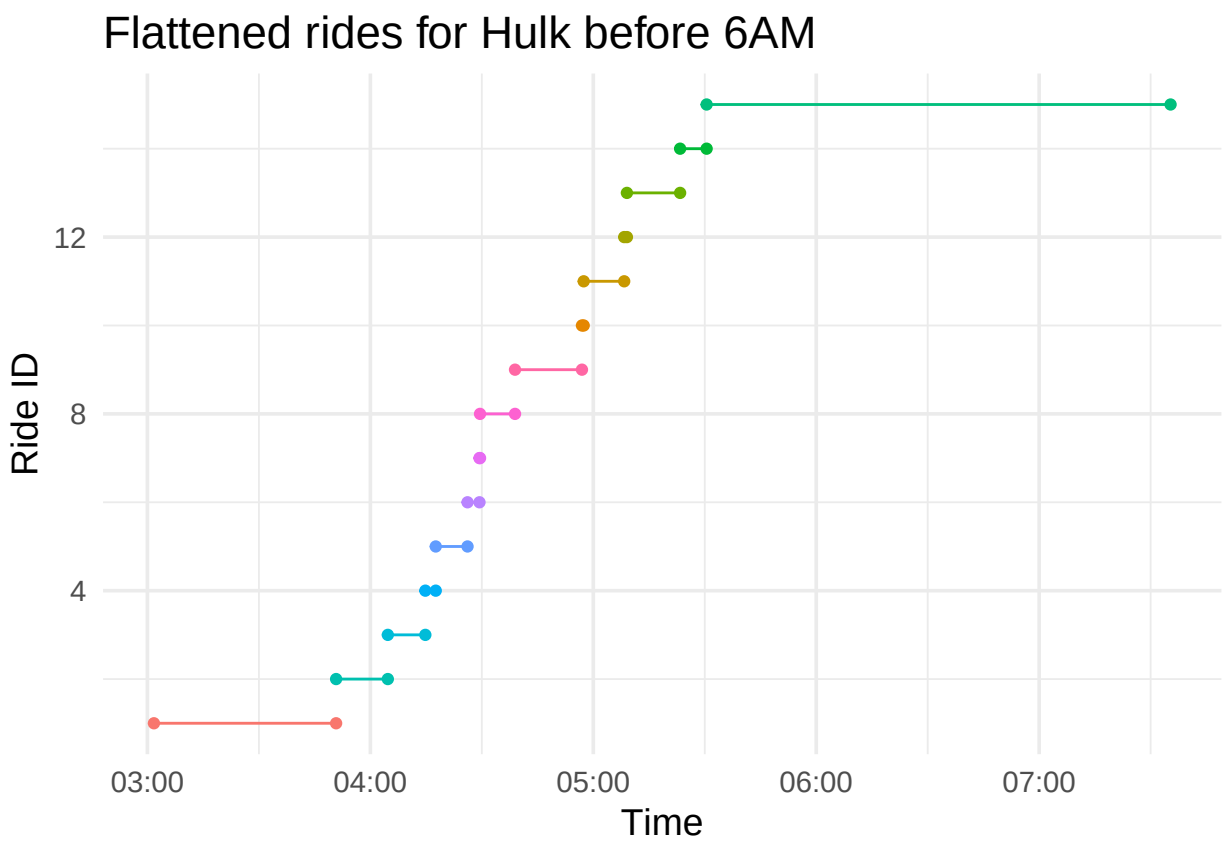
```
# select only necessary items and sort by car ID
df_u <- df %>%
  select(car_id, start_time, end_time, ride_mins, num_riders) %>%
  arrange(car_id)

# create vector lists of each feature to prepare for transformation in python
car_id <- df_u$car_id
start_time <- period_to_seconds(seconds(df_u$start_time))
end_time <- period_to_seconds(seconds(df_u$end_time))
ride_mins <- df_u$ride_mins
num_riders <- as.integer(df_u$num_riders)
```

```
# create an array of rides dictionaries
rides = []
for c, s, e, m, n in zip(r.car_id, r.start_time, r.end_time, r.ride_mins, r.num_riders):
  rides.append({
    "car_id": c,
    "start_time": s,
    "end_time": e,
    "ride_mins": m,
    "num_riders": n
  })
# create a function to find weighted average riders of a block of rides
def avg_num_riders(rides):
  weighted_riders = 0
  total_mins = 0
  for i in range(len(rides)):
    #print(rides[i]["ride_mins"], rides[i]["num_riders"])
    weighted_riders += rides[i]["ride_mins"] * rides[i]["num_riders"]
    total_mins += rides[i]["ride_mins"]
  average_riders = weighted_riders / total_mins
```

```
  #print(average_riders)
  return average_riders

# iterate over rides array to build a new array with no overlap
rides_flattened = []
ride_sequence = [rides[0]]
earliest_start_time = rides[0]["start_time"]
latest_end_time = rides[0]["end_time"]
for i in range(len(rides)-1):
  if rides[i+1]["start_time"] < latest_end_time \
  and rides[i+1]["car_id"] == rides[i]["car_id"]:
    ride_sequence.append(rides[i+1])
    latest_end_time = max(latest_end_time, rides[i+1]["end_time"])
  else:
    avg_riders_for_sequence = avg_num_riders(ride_sequence)
    rides_flattened.append({
      "car_id": rides[i]["car_id"],
      "start_time": earliest_start_time,
      "end_time": latest_end_time,
      "num_riders": avg_riders_for_sequence
    })
    if rides[i+1]["car_id"] == rides[i]["car_id"]:
      rides_flattened.append({
        "car_id": rides[i]["car_id"],
        "start_time": latest_end_time,
        "end_time": rides[i+1]["start_time"],
        "num_riders": 0.0
      })
    earliest_start_time = rides[i+1]["start_time"]
    latest_end_time = rides[i+1]["end_time"]
    ride_sequence = [rides[i+1]]
# create individual lists to transform back into an R dataframe
car_id_flattened = []
start_time_flattened = []
end_time_flattened = []
num_riders_flattened = []
for i in range(len(rides_flattened)):
  car_id_flattened.append(rides_flattened[i]["car_id"])
  start_time_flattened.append(rides_flattened[i]["start_time"])
  end_time_flattened.append(rides_flattened[i]["end_time"])
  num_riders_flattened.append(rides_flattened[i]["num_riders"])
```

```
# convert Python lists back into R lists
start_time_flattened_POSXct = as.POSIXct(py$start_time_flattened,
                                         tz="UTC",
                                         origin = "1970-01-01")
end_time_flattened_POSXct = as.POSIXct(py$end_time_flattened,
                                       tz="UTC",
                                       origin = "1970-01-01")

# build an R data frame from the lists
df_uf <- data.frame("car_id" = py$car_id_flattened,
                    "start_time" = start_time_flattened_POSXct,
```

```
                    "end_time" = end_time_flattened_POSXct,
                    "num_riders" = py$num_riders_flattened) %>%
  mutate(ride_mins = as.double(end_time - start_time)/60)

# filter to one car for a small period of time
df_hulk2 <- df_uf %>%
  filter(car_id == "hulk") %>%
  filter(start_time < ymd_hms("2018-10-02 6:00:00")) %>%
  mutate(ride_id = row_number()) %>%
  select(ride_id, start_time, end_time, ride_mins, num_riders)

# visualize a few rides for a single car
df_hulk2 %>%
  gather("startORend", "time", 2:3) %>%
  arrange(ride_id) %>%
  ggplot(aes(time, ride_id, color=as.character(ride_id))) +
  geom_point() +
  geom_line() +
  guides(color=FALSE) +
  labs(x="Time",
     y="Ride ID",
     title="Flattened rides for Hulk before 6AM")
```

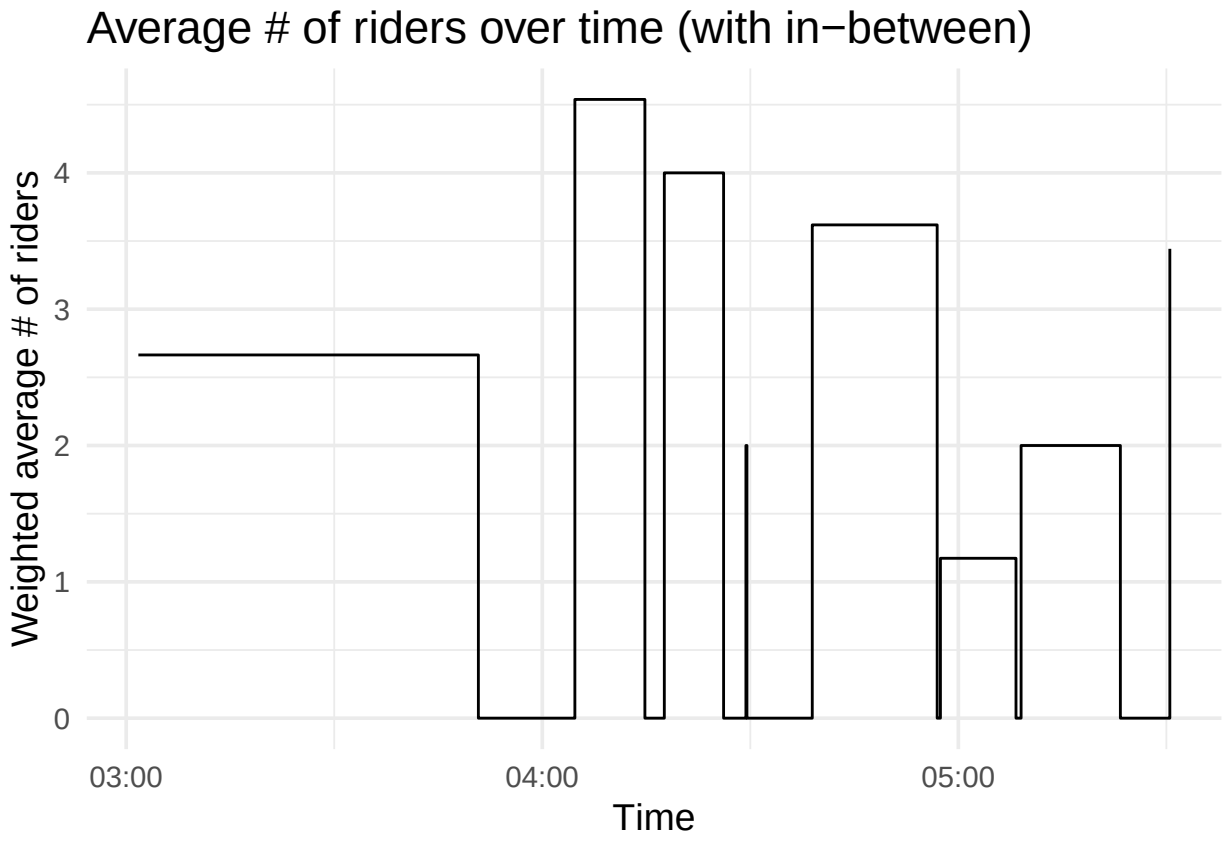



```
# print table of first 5 rides
kable(head(mutate(df_hulk2, ride_mins = round(ride_mins))), booktabs = TRUE,
      caption="Trips are now flattened with a average # riders")
```

Table 4: Trips are now flattened with a average # riders

ride_id	start_time	end_time	ride_mins	num_riders
1	2018-10-02 03:01:45	2018-10-02 03:50:47	49	2.664330
2	2018-10-02 03:50:47	2018-10-02 04:04:42	14	0.000000
3	2018-10-02 04:04:42	2018-10-02 04:14:48	10	4.538686
4	2018-10-02 04:14:48	2018-10-02 04:17:36	3	0.000000
5	2018-10-02 04:17:36	2018-10-02 04:26:09	9	4.000000
6	2018-10-02 04:26:09	2018-10-02 04:29:22	3	0.000000

```
# visualize the Hulk average number of riders for a limited time period
ggplot(df_hulk2, aes(start_time, num_riders)) +
  geom_step() +
  labs(x="Time",
     y="Weighted average # of riders",
     title="Average # of riders over time (with in-between)")
```



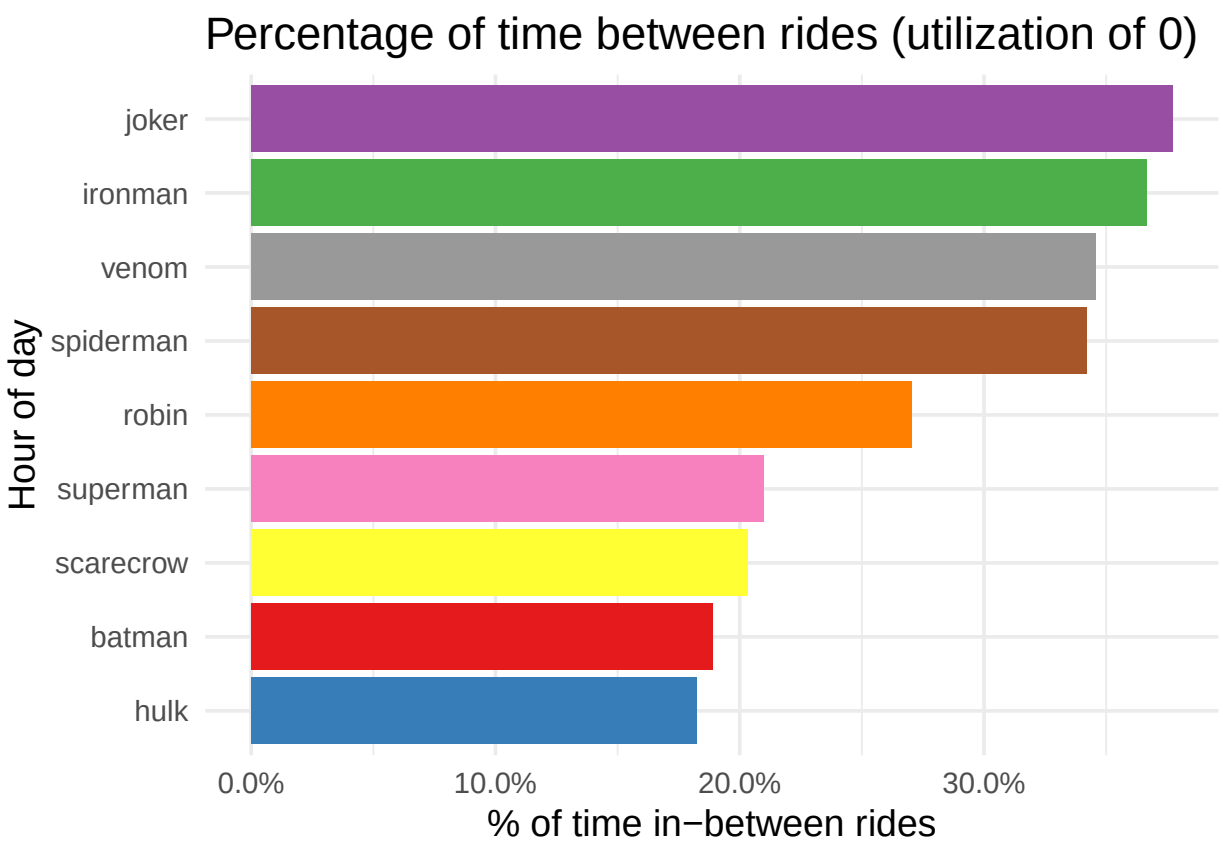
Algorithmic outcome

It can be seen that the Python algorithm worked as expected, and it outputted a flattened list or rides over time, including in-between times. This flattened list can now be used to accurately generate a number of utilization metrics and visulizations. The average utilization over time is 48% (keeping in mind that this is by the total number of seats).

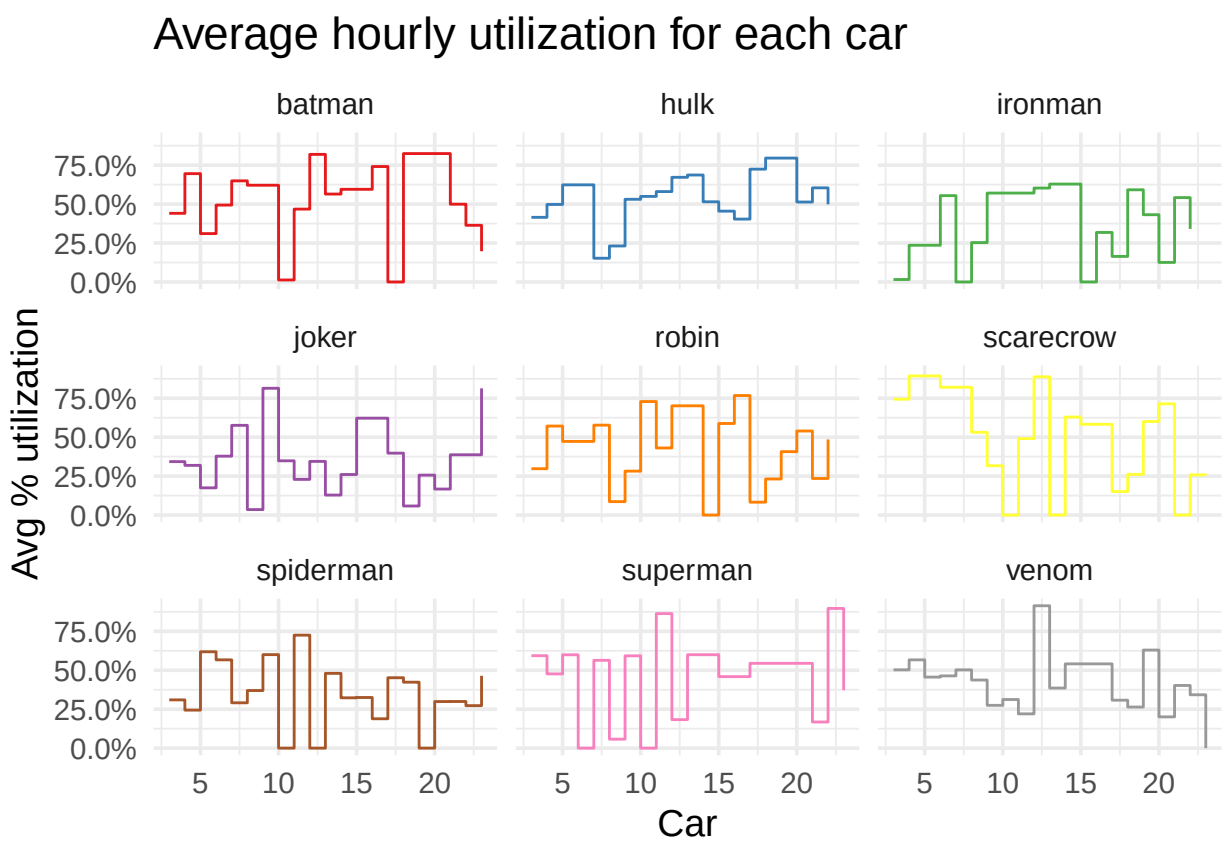
Results and recommendations

Utilization charts:

```
overall_average_utilization <- df_uf %>%
  summarize(((sum(num_riders * ride_mins)) / (sum(ride_mins)))/5) %>%
  as.double()

# average utilization per car
df_uf_car <- df_uf %>%
  group_by(car_id) %>%
  summarize(avg_num_riders = (sum(num_riders * ride_mins)) / (sum(ride_mins)),
            avg_utilization = avg_num_riders / 5,
            percent_time_between_rides = sum(ride_mins[num_riders==0])/sum(ride_mins))

# visualize time between rides (wasted time with 0 people)
ggplot(df_uf_car, aes(fct_reorder(car_id, percent_time_between_rides), percent_time_between_rides, fill=
  geom_bar(stat="identity") +
  coord_flip() +
  scale_fill_brewer(palette="Set1") +
  guides(fill=FALSE) +
  scale_y_continuous(labels = percent) +
  labs(x="Hour of day",
       y="% of time in-between rides",
       title="Percentage of time between rides (utilization of 0)")
```

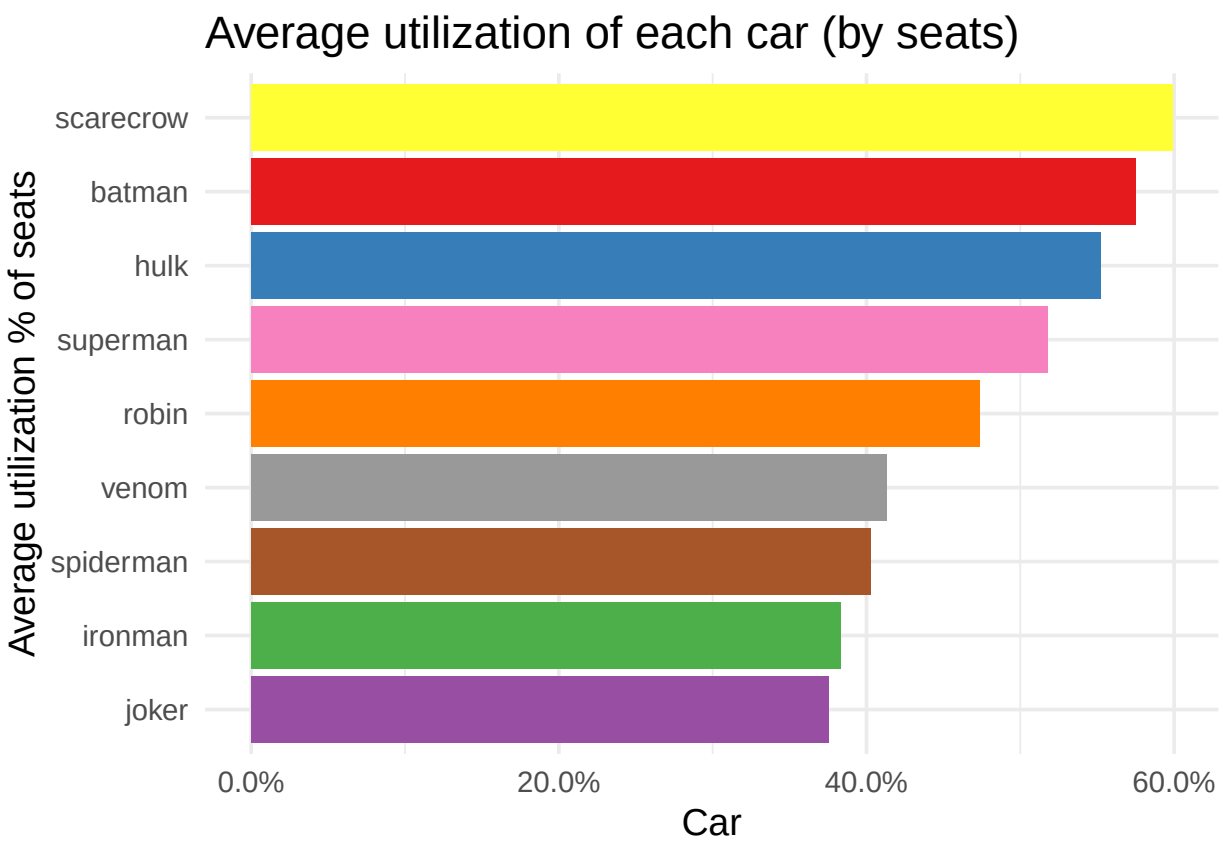


```
# average utilization per hour per car (based on total # of seats available)
df_uf %>% mutate(hour = hour(start_time)) %>%
  group_by(car_id, hour) %>%
  summarize(avg_num_riders = (sum(num_riders * ride_mins)) / (sum(ride_mins)),
            avg_utilization = avg_num_riders / 5) %>%
  ungroup() %>%
  ggplot(aes(hour, avg_utilization, color=car_id)) +
    geom_step() +
    facet_wrap(~car_id) +
    scale_color_brewer(palette="Set1") +
    guides(color=FALSE) +
    scale_y_continuous(labels = percent) +
    labs(x="Car",
         y="Avg % utilization",
         title="Average hourly utilization for each car")
```




```
# average utilization per
ggplot(df_uf_car, aes(fct_reorder(car_id, avg_utilization),
                      avg_utilization,
                      fill=car_id)) +
  geom_bar(stat="identity") +
  coord_flip() +
  scale_fill_brewer(palette="Set1") +
  guides(fill=FALSE) +
  scale_y_continuous(labels = percent) +
  labs(x="Average utilization % of seats",
       y="Car",
```

```
      title="Average utilization of each car (by seats)")
```



Business actions:

What originally seemed like high utilization turned out to actually be a lot of empty seats once we restructured the data. There are no definitive trends in utilization over time, nor is any one car significantly higher than any other. Scarecrow had the highest utilization rate, at about 60%; and Joker had the lowest, at about 38%. There is also quite a lot of lost time in-between rides, with 38% of its time with zero passengers and with the Hulk only spending 18% of its time inbetween passengers. Overall, this means that there is clearly room for route optimization, and for better understanding how many people will be getting into a car any any given time in order to maximize utilization. As we'll see in the next section, improving utilization makes a much bigger impact if pricing is calculated more dynamically and intelligently.

Pricing

Pricing is one variable that businesses can exercise some degree of control over, and it has big repurcussions on many business dynamics and importantly, the bottom line. I've actually written a post on frameworks for thinking about pricing, which might be useful for future analyses and pricing decisions.

Some notable findings:

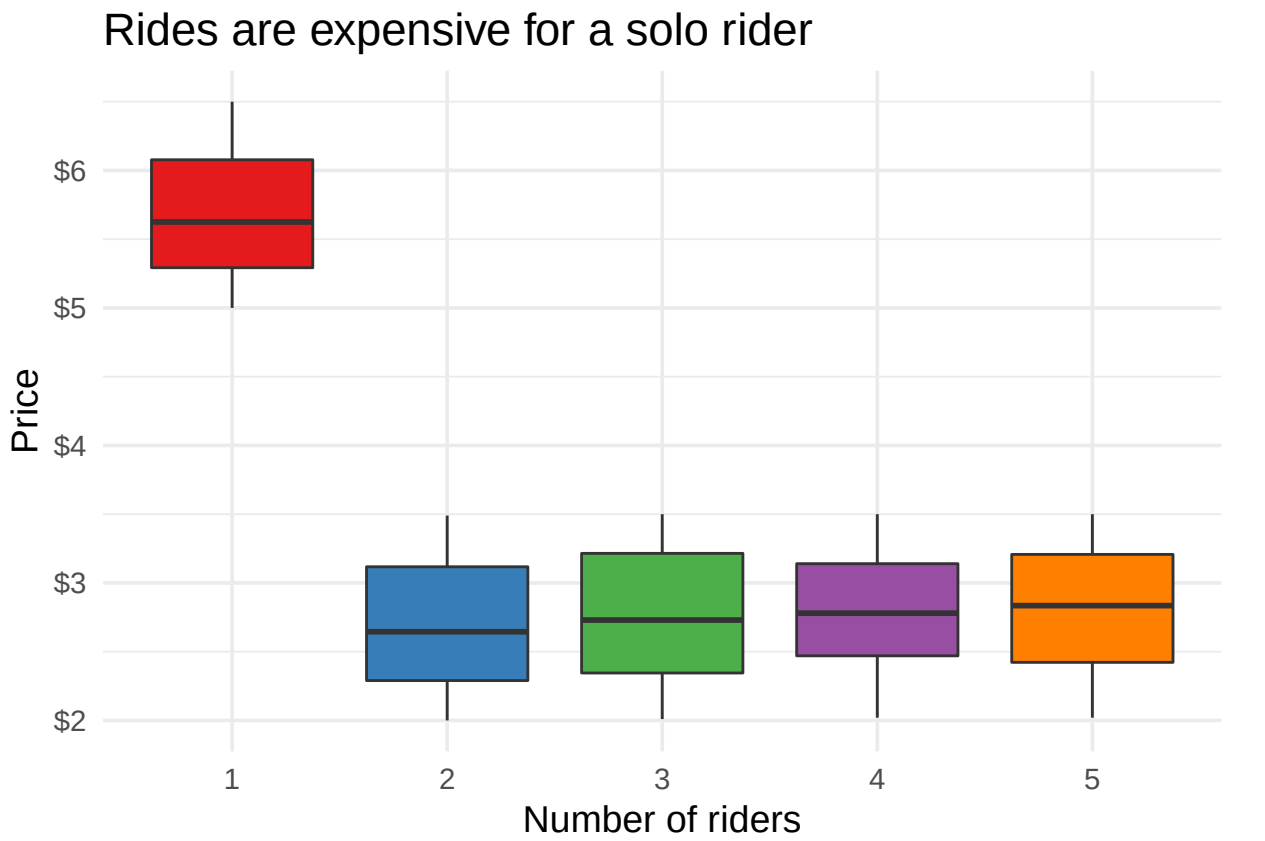
- There is no correlation (-0.0543) between price and ride duration (reference first chart in this report)
- Rides are always "expensive" (>$5) when there is 1 rider

- Rides are always "cheap" (<$3.5) when there is more than 1 rider; there is no increase in price for additional riders from 2-5

Charts to demonstrate:

*Note: I am assuming that "price" is for the ride, and not for each individual in the ride. If it were the latter (price is per person in the car), then calculations and suggested actions would be slightly different. Although they wouldn't change all that much, since each car had relatively the same number of total riders (with the exception of the hulk, which had 504 instead of an average of 330)

```
avg_total_riders_not_hulk <- df %>%
  filter(car_id!="hulk") %>%
  group_by(car_id) %>%
  summarize(total_riders = sum(as.integer(num_riders))) %>%
  ungroup() %>%
  summarize(mean(total_riders)) %>% as.double

ggplot(df, aes(num_riders, price, fill = num_riders)) +
  geom_boxplot() +
  scale_fill_brewer(palette = "Set1") +
  guides(fill=FALSE) +
  scale_y_continuous(labels = dollar) +
  labs(x="Number of riders",
       y="Price",
       title="Rides are expensive for a solo rider")
```



Business actions:

My biggest recommendation would be to calculate pricing dynamically based on both (1) the expected duration of the ride and (2) the number of riders. This would be a good place to start that would have a significant impact on your total revenues. This is about all the pricing recommendations we can get from the data set, but there are a number of things that Ryde Automation can do to improve dynamic pricing. With a larger data set, I would guess that trends would emerge about utilization changing based on the time of day, the day of the week, or even the season, all of which could be used as features in a dynamic pricing algorithm. More generally, there are numerous other features that could be used to improve pricing. One particularly powerful technique is better understanding specific customers', or customer segments', willingness to pay. When a potential customer opens the app, looks at the price, and then closes the app, the price is beyond their threshold. With more data points, a demand curve can be built, enabling an incredible opportunity for price optimization.