Project: Bicycle Sharing

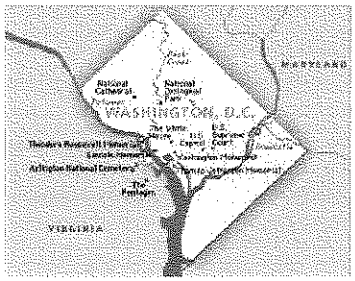

Figure A.14: Washington, D.C.

Capital BikeShare is a bicycle-sharing system in Washington, D.C. At any of about 400 stations, a registered user can unlock and check out a bicycle. After use, the bike can be returned to the same station or any of the other stations.

Such sharing systems require careful management. There need to be enough bikes at each station to satisfy the demand, and enough empty docks at the destination station so that the bikes can be returned. At each station, bikes are checked out and are returned over the course of the day. An imbalance between bikes checked out and bikes returned calls for the system administration to truck bikes from one station to another. This is expensive.

In order to manage the system, and to support a smart-phone app so that users can find out when bikes are available, Capital BikeShare collects real-time data on when and where each bike is checked out or returned, how many bikes and empty docks there are at each station. Capital BikeShare publishes the station-level information in real time. The organization also publishes, at the end of each quarter of the year, the historical record of each bike rental in that time.

You can access the data from the Capital Bikeshare web site. Doing this requires some translation and cleaning, skills that are introduced in Chapter (15). For this project, however, already translated and cleaned data are provided in the form of two data tables:

- `Stations` giving information about location of each of the stations in the system.

  ```
  Stations <- mosaic::read.file("http://tiny.cc/dcf/DC-Stations.csv")
  ```

- `Trips` giving the rental history over the last quarter of 2014.

  ```
  data_site <- "http://tiny.cc/dcf/2014-Q4-Trips-History-Data-Small.rds"
  Trips <- readRDS(gzcon(url(data_site)))
  ```

  The `Trips` data table is a random subset of 10,000 trips from the full quarterly data. Start with this small data table to develop your analysis commands. When you have this working well, you can

access the full data set of more than 600,000 events by removing `-Small` from the name of the `data_site`.

In this activity, you'll work with just a few variables:

- From `Stations`:
  - the latitude and longitude of the station (`lat` and `long`, respectively)
  - `name`: the station's name
- From `Trips`:
  - `sstation`: the name of the station where the bicycle was checked out.
  - `estation`: the name of the station to which the bicycle was returned.
  - `client`: indicates whether the customer is a `"regular"` user who has paid a yearly membership fee, or a `"casual"` user who has paid a fee for five-day membership.
  - `sdate`: the time and date of check-out
  - `edata`: the time and date of return

Notice that the location and time variable start with an "s" or an "e" to indicate whether the variable is about the start of a trip or the end of a trip.

Time/dates are stored as a special kind of number: a *POSIX date*. You can use `sdate` and `edate` in the same way that you would use a number. For instance, Figure A.15 shows the distribution of times that bikes were checked out.

POSIX DATE: A representation of date and time of day that facilitates using dates in the same way as numbers, e.g. to find the time elapsed between two dates.

```
Trips %>%
  ggplot(aes(x=sdate)) +
  geom_density(fill="gray", color=NA)
```

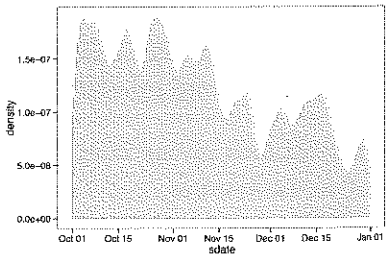

Figure A.15: Use of shared bicycles over the three months in Q3.

A.1 How long?

Your Turn: Make a box-and-whisker plot, like Figure A.16 showing the distribution of the duration of rental events, broken down by the `client` type. The duration of the rental can be calculated as `as.numeric(edate - sdate)`. The units will be in either hours, minutes, or seconds. You figure it out.

When you make your plot, you will likely find that the axis range is being set by a few outliers. These may be bikes that were forgotten. Arrange your scale to ignore these outliers, or filter them out.

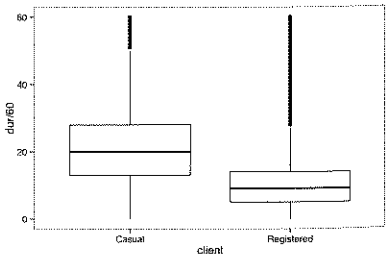

Figure A.16: The distribution of bike-rental durations as a box-and-whisker plot.

A.2 *When are bikes used?*

The `sdate` variable in `Trips` indicates the date and time of day that the bicycle was checked out of the parking station. `sdate` is stored as a special

Often, you will want discrete components of a date, for instance:

- The day of the year (1 to 365): use `lubridate:yday(sdate)`
- The day of the week (Sunday to Saturday): use `lubridate::wday(sdate)`
- The hour of the day: use `lubridate::hour(sdate)`
- The minute in the hour: use `lubridate::minute(sdate)`

Your Turn: Make histograms or density plots of each of these discrete components. Explain what each plot is showing about how bikes are checked out. For example, Figure A.17 shows that few bikes are checked out before 5am, and that there are busy times around the rush hour: 8am and 5pm.

```
Trips %>%
  mutate(H = lubridate::hour(sdate)) %>%
  ggplot(aes(x = H)) +
  geom_density(fill="gray", adjust=2)
```

A similar sort of display of events per hour can be accomplished by calculating and displaying each hour's count, as in Figure A.18.

```
Trips %>%
  mutate(H = lubridate::hour(sdate)) %>%
  group_by(H) %>%
  summarise(count = n()) %>%
  ggplot(aes(x=H, y=count)) +
    geom_point() + geom_line()
```

The graphic shows a lot of variation of bike use over the course of the day. Now consider two additional variables: the day of the week and the "client" type.

Your Turn: Group the bike rentals by three variables: hour of the day, day of the week, and client type. Find the total number of events in each grouping and plot this count versus hour. Use the `group` aesthetic to represent one of the other variables and faceting to represent the other. (Recall that faceting is done by adding `facet_wrap( ~var)` to the plotting commands.)

Your Turn (optional): Make the same sort of display of how bike rental vary of hour, day of the week, and client type, but use `geom_density()` rather than grouping and counting. Compare the two displays — one of discrete counts and one of density — and describe any major differences.

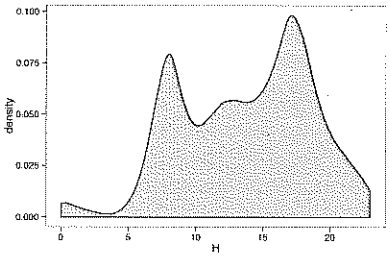

Figure A.17: Distribution of bike trips by hour of the day.

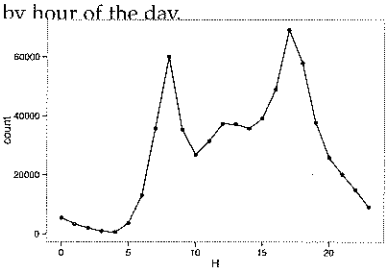

Figure A.18: The number of events in each hour of the day. Compare to the scale in Fig. effig:bikes-over-hours. Why are the scales different?

A.3 *How far?*

Find the distance between each pair of stations. You know the position from the `lat` and `long` variables in `Stations`. This is enough information to find the distance. The calculation has been implemented in the `haversine()` function:

```
source("http://tiny.cc/dcf/haversine.R")
```

`haversine()` is a transformation function. To use it, create a data table where a case is a *pair* of stations and there are variables for the latitude and longitude of the starting station and the ending station. To do this, join the `Station` data to itself. The following statements show how to create appropriately named variables for joining.

```
Simple <-
  Stations %>%
  select(name, lat, long) %>%
  rename(sstation=name)
Simple2 <-
  Simple %>%
  rename(estation=sstation, lat2=lat, long2=long)
```

Look at the `head()` of `Simple` and `Simple2` and make sure you understand how they are related to `Stations`.

The joining of `Simple` and `Simple2` should match every station to every other station. This sort of matching does not make use of any matching variables; everything is matched to everything else. This is called a *full outer join*.

Since a ride can start and end at the same station, it also makes sense to match each station to itself.

FULL OUTER JOIN: A join which matches every case in the left table to each and every case in the right table.

First, try the full outer join of just a few cases from each table, for example 4 from the left and 3 from the right.

```
merge(head(Simple, 4), head(Simple2, 3), by=NULL)
```

Make sure you understand what the full outer join does before proceeding. For instance, you should be able to predict how many cases the output will have when the left input has n cases and the right has m cases.

- There are 347 cases in the `Stations` data table. How many cases will there be in a full outer join of `Simple` to `Simple2`?

It's often impractical to carry out a full outer join. For example, joining `BabyNames` to itself with a full outer join will generate a result with more than three-trillion cases.

There are only a few computers in the world that are be able to hold three trillion cases from `BabyNames`. It's the equivalent of about 5 million hours of MP3 compressed music. A typical human lifespan is about 0.6 million hours.

Perform the full outer join and then use `haversine()` to compute the distance between each pair of stations.

```
StationPairs <- merge(Simple, Simple2, by=NULL)
```

Check your result for sensibility. Make a histogram of the station-to-station distances and explain where it looks like what you would expect. (Hint: from one end of Washington, D.C. to the other is about 14.1 miles.)

```
PairDistances <-
  StationPairs %>%
  mutate(distance = haversine(lat, long, lat2, long2)) %>%
  select(sstation, estation, distance)
```

Once you have `PairDistances`, you can join it with `Trips` to calculate the start-to-end distance of each trip.

Of course, a rider may not go directly from one station to another.

- Look at the variables in `Stations` and `Trips` and explain why `Simple` and `Simple2` were given different variable names for the station.

An `inner_join()` is appropriate for finding the distance of each ride.

Watch out! `Trips` and `PairDistances` are large enough that the join is expensive: it takes about a minute.

```
RideDists <-
  Trips %>%
  inner_join(PairDistances)
```

Your turn: Display the distribution of the ride distances of the rides. Compare it to the distances between pairs of stations. Are they similar? Why or why not?

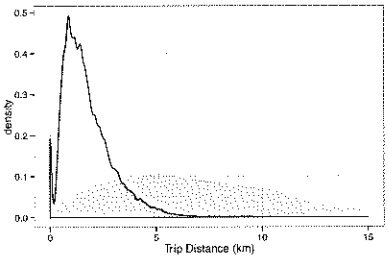

Figure A.19: The distribution of trip lengths compared to the distribution of distances between pairs of stations (shaded).

A.4 Mapping the Stations

You can draw detailed maps with the `leaflet` package. You may need to install it.

```
devtools::install_github("rstudio/leaflet")
```

`leaflet` works much like `ggplot()` but provides special facilities for maps. Here's how to make the simple map:

```
library(leaflet)
stationMap <-
  leaflet(Stations) %>%    # like ggplot()
  addTiles() %>%           # add the map
  addCircleMarkers(radius=2, color="red") %>%
  setView(-77.04, 38.9, zoom=12)
```

To display the map, use the object name as a command: `stationMap`

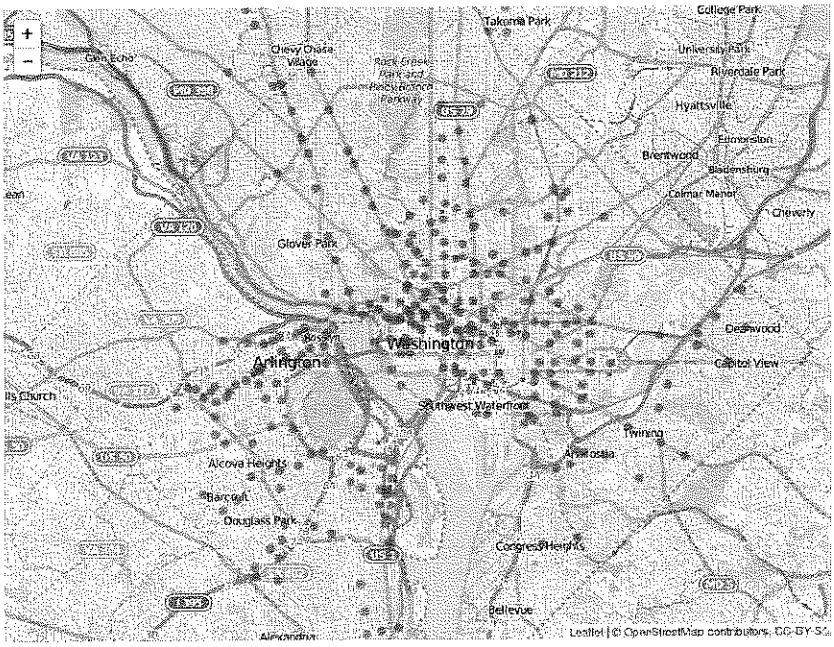

Figure A.20: A map of station locations drawn with the `leaflet()` function in the `leaflet` package.

A.5 Long-distance stations

Your Turn (optional, but cool): Around each station on the map, draw a circle whose radius reflects the median distance covered by rentals starting at that station. To draw the circles, use the same `leaflet` commands as before, but add in a line like this.

```
addCircles(radius = ~ mid, color="blue", opacity=0.001)
```

For `addCircles()` to draw circles at the right scale, the units of the median distance should be presented in meters rather than kilometers. This will create too much overlap, unfortunately. So, set the radius to be half or one-third the median distance in meters. From your map, explain the pattern you see in the relationship between station location and median distance.