Advanced Computing 1 – Data wrangling and plotting

Stefano Allesina

Data Wrangling and Plotting

- **Goal:** learn how to manipulate large data sets by writing efficient, consistent, and compact code. Introduce the use of `dplyr`, `tidyr`, and the "pipeline" operator `%>%`. Produce beautiful graphs and figures for scientific publications using `ggplot2`.
- **Audience:** experienced `R` users, familiar with the data type `data.frame`, loops, functions, and having some notions of data bases.
- **Installation:** the following packages need to be installed: `ggplot2`, `dplyr`, `tidyr`, `lubridate`, `ggthemes`

Data wrangling

As biologists living in the XXI century, we are often faced with tons of data, possibly replicated over several organisms, treatments, or locations. We would like to streamline and automate our analysis as much as possible, writing scripts that are easy to read, fast to run, and easy to debug. Base `R` can get the job done, but often the code contains complicated operations (think of the cases in which you used `lapply` only because of its speed), and a lot of `$` signs and brackets.

We're going to learn about the packages `dplyr` and `tidyr`, which can be used to manipulate large data frames in a simple and straightforward way. These tools are also much faster than the corresponding base `R` commands, are very compact, and can be concatenated into "pipelines".

To start, we need to import the libraries:

```
library(dplyr)
library(tidyr)
```

Then, we need a dataset to play with. We take a dataset containing all the Divvy bikes trips in Chicago in July 2014:

```
divvy <- read.csv("../data/Divvy_Trips_July_2014.csv")
```

A new data type, `tbl`

This is now a data frame:

```
is.data.frame(divvy)
```

`dplyr` ships with a new data type, called a `tbl`. To convert from data frame, use

```
divvy <- tbl_df(divvy)
divvy
```

The nice feature of `tbl` objects is that they will print only what fits on the screen, and also give you useful information on the size of the data, as well as the type of data in each column. Other than that, a `tbl` object behaves very much like a `data.frame`. In some rare cases, you want to transform the `tbl` back into a `data.frame`. For this, use the function `as.data.frame(tbl_object)`.

We can take a look at the data using one of several functions:

- `head(divvy)` shows the first few (10 by default) rows
- `tail(divvy)` shows the last few (10 by default) rows
- `glimpse(divvy)` a summary of the data (similar to `str` in base R)
- `View(divvy)` open in spreadsheet-like window

Selecting rows and columns

There are many ways to subset the data, either by row (subsetting the *observations*), or by column (subsetting the *variables*). For example, suppose we want to count how many trips (of the > 410k) are very short. The column `tripduration` contains the length of the trip in seconds. Let's select only the trips that lasted less than 3 minutes:

```
filter(divvy, tripduration < 180)
```

You can see that "only" 11,099 trips lasted less than three minutes. We have used the command `filter(tbl, conditions)` to select certain observations. We can combine several conditions, by listing them side by side, possibly using logical operators.

> **Exercise:** what does this do?
>
> ```
> filter(divvy, gender == "Male", tripduration > 60, tripduration < 180)
> ```

We can also select particular variables using the function `select(tbl, cols to select)`. For example, select `from_station_name` and `from_station_id`:

```
select(divvy, from_station_name, from_station_id)
```

How many stations are represented in the data set? We can use the function `distinct(tbl)` to retain only the rows that differ from each other:

```
distinct(select(divvy, from_station_name, from_station_id))
```

Showing that there are 300 stations, once we removed the duplicates.

Other ways to subset observations:

- `sample_n(tbl, howmany, replace = TRUE)` sample `howmany` rows at random with replacement
- `sample_frac(tbl, proportion, replace = FALSE)` sample a certain proportion (e.g. `0.2` for 20%) of rows at random without replacement
- `slice(tbl, 50:100)` extract the rows between `50` and `100`
- `top_n(tbl, 10, tripduration)` extract the first `10` rows, once ordered by `tripduration`

More ways to select columns:

- `select(divvy, contains("station"))` select all columns containing the word `station`
- `select(divvy, -gender, -tripduration)` exclude the columns `gender` and `tripduration`
- `select(divvy, matches("year|time"))` select all columns whose names match a regular expression

Creating pipelines using %>%

We've been calling nested functions, such as `distinct(select(divvy, ...))`. If you have to add another layer or two, the code would become unreadable. `dplyr` allows you to "un-nest" these functions and create a "pipeline", in which you concatenate commands separated by the special operator `%>%`. For example:

```
divvy %>% # take a data table
  select(from_station_name, from_station_id) %>% # select two columns
  distinct() # remove duplicates
```

does exactly the same as the command above, but is much more readable. By concatenating many commands, you can create incredibly complex pipelines while retaining readability.

Producing summaries

Sometimes we need to calculate statistics on certain columns. For example, calculate the average trip duration. We can do this using `summarise`:

```
divvy %>% summarise(avg = mean(tripduration))
```

which returns a `tbl` object with just the average trip duration. You can combine multiple statistics (use `first`, `last`, `min`, `max`, `n` [count the number of rows], `n_distinct` [count the number of distinct rows], `mean`, `median`, `var`, `sd`, etc.):

```
divvy %>% summarise(avg = mean(tripduration),
                    sd = sd(tripduration),
                    median = median(tripduration))
```

Summaries by group

One of the most useful features of `dplyr` is the ability to produce statistics for the data once subsetted by *groups*. For example, we would like to measure whether men take longer trips than women. We can then group the data by `gender`, and calculate the mean `tripduration` once the data is split into groups:

```
divvy %>% group_by(gender) %>% summarise(mean = mean(tripduration))
```

showing that women tend to take longer trips than men.

> **Exercise:** count the number of trips for Male, Female, and unspecified gender.

Ordering the data

To order the data according to one or more variables, use `arrange()`:

```
divvy %>% select(trip_id, tripduration) %>% arrange(tripduration)
divvy %>% select(trip_id, tripduration) %>% arrange(desc(tripduration))
```

Renaming columns

To rename one or more columns, use `rename()`:

```
divvy %>% rename(tt = tripduration)
```

Adding new variables using mutate

If you want to add one or more new columns, use the function `mutate`:

```
divvy %>% select(from_station_id, to_station_id) %>%
  mutate(mylink = paste0(from_station_id, "->", to_station_id))
```

use the function `transmute()` to create a new column and drop the original columns. You can also use `mutate` and `transmute` on grouped data:

```
# A more complex pipeline
divvy %>%
  select(trip_id, gender, tripduration) %>%  # select only three columns
  rename(t = tripduration) %>% # rename a column
  group_by(gender) %>% # create a group for each gender value
  mutate(zscore = (t - mean(t) ) / sd(t)) %>% # compute z-score for t, according to gender
  ungroup() %>% # remove group information
  arrange(desc(t), zscore, gender) %>% # order by t (decreasing), zscore, and gender
  head(20) # display first 20 rows
```

Data plotting

The most salient feature of scientific graphs should be clarity. Each figure should make crystal-clear a) what is being plotted; b) what are the axes; c) what do colors, shapes, and sizes represent; d) the message the figure wants to convey. Each figure is accompanied by a (sometimes long) caption, where the details can be explained further, but the main message should be clear from glancing at the figure (often, figures are the first thing editors and referees look at).

Many scientific publications contain very poor graphics: labels are missing, scales are unintelligible, there is no explanation of some graphical elements. Moreover, some color graphs are impossible to understand if printed in black and white, or difficult to discern for color-blind people.

Given the effort that you put in your science, you want to ensure that it is well presented and accessible. The investment to master some plotting software will be rewarded by pleasing graphics that convey a clear message.

In this section, we introduce `ggplot2`, a plotting package for `R` This package was developed by Hadley Wickham who contributed many important packages to `R` (including `dplyr`). Unlike many other plotting systems, `ggplot2` is deeply rooted in a "philosophical" vision. The goal is to conceive a grammar for all graphical representation of data. Leland Wilkinson and collaborators proposed The Grammar of Graphics. It follows the idea of a well-formed sentence that is composed of a subject, a predicate, and an object. The Grammar of Graphics likewise aims at describing a well-formed graph by a grammar that captures a very wide range of statistical and scientific graphics. This might be more clear with an example – Take a simple two-dimensional scatterplot. How can we describe it? We have:

- **Data** The data we want to plot.
- **Mapping** What part of the data is associated with a particular visual feature? For example: Which column is associated with the x-axis? Which with the y-axis? Which column corresponds to the shape or the color of the points? In `ggplot2` lingo, these are called *aesthetic mappings* (`aes`).
- **Geometry** Do we want to draw points? Lines? In `ggplot2` we speak of *geometries* (`geom`).
- **Scale** Do we want the sizes and shapes of the points to scale according to some value? Linearly? Logarithmically? Which palette of colors do we want to use?
- **Coordinate** We need to choose a coordinate system (e.g., Cartesian, polar).
- **Faceting** Do we want to produce different panels, partitioning the data according to one (or more) of the variables?

This basic grammar can be extended by adding statistical transformations of the data (e.g., regression, smoothing), multiple layers, adjustment of position (e.g., stack bars instead of plotting them side-by-side), annotations, and so on.

Exactly like in the grammar of a natural language, we can easily change the meaning of a "sentence" by adding or removing parts. Also, it is very easy to completely change the type of geometry if we are moving from say a histogram to a boxplot or a violin plot, as these types of plots are meant to describe one-dimensional distributions. Similarly, we can go from points to lines, changing one "word" in our code. Finally, the look and feel of the graphs is controlled by a theming system, separating the content from the presentation.

Basic `ggplot2`

`ggplot2` ships with a simplified graphing function, called `qplot`. In this introduction we are not going to use it, and we concentrate instead on the function `ggplot`, which gives you complete control over your plotting. First, we need to load the package. While we are at it, let's also load a package extending its theming system:

```
library(ggplot2)
library(ggthemes)
```

And then, let's get a small data set, containing the data on the Divvy stations:

```
divvy_stations <- read.csv("../data/Divvy_Stations_July_2014.csv")
```

A particularity of `ggplot2` is that it accepts exclusively data organized in tables (a `data.frame` or a `tbl` object). Thus, all of your data needs to be converted into a data frame format for plotting.

Let's look at the data:

```
head(divvy_stations)
```

For our first plot, we're going to plot the position of the stations, using the latitude (y axis) and longitude (x axis). First, we need to specify a dataset to use:

```
ggplot(data = divvy_stations)
```

As you can see, nothing is drawn: we need to specify what we would like to associate to the x axis, and what to the y axis (i.e., we want to set the *aesthetic mappings*):

```
ggplot(data = divvy_stations) + aes(x = longitude, y = latitude)
```

Note that we concatenate pieces of our "sentence" using the `+` sign! We've got the axes, but still no graph... we need to specify a geometry. Let's use points:

```
ggplot(data = divvy_stations) + aes(x = longitude, y = latitude) + geom_point()
```

You can now see the outline of Chicago, with the lake on the right (east), the river separating the Loop from the West Loop, etc. As you can see, we wrote a well-formed sentence, composed of **data** + **mapping** + **geometry**. We can add other mappings, for example, showing the capacity of the station using different point sizes:

```
ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, size = dpcapacity) +
  geom_point()
```

or colors

```
ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, colour = dpcapacity) +
  geom_point()
```

Scatterplots

Using `ggplot2`, one can produce very many types of graphs. The package works very well for 2D graphs (or 3D rendered in two dimensions), while it lack capabilities to draw proper 3D graphs, or networks.

The main feature of `ggplot2` is that you can tinker with your graph fairly easily, and with a common grammar. You don't have to settle on a certain presentation of the data until you're ready, and it is very easy to switch from one type of graph to another.

For example, let's calculate the median `tripduration` by `birthdate`, to see whether older people tend to take longer or shorter trips:

```
duration_byyr <- divvy %>%
  filter(is.na(birthyear) == FALSE) %>% # remove records without birthdate
  filter(birthyear > 1925) %>% # remove ultra centenarian people (probably, errors)
  group_by(birthyear) %>% # group by birth year
  summarise(median_duration  = median(tripduration)) # calculate median for each group

pl <- ggplot(data = duration_byyr) + # data
  aes(x = birthyear, y = median_duration) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

We can add a smoother by typing

```
pl + geom_smooth() # spline by default
pl + geom_smooth(method = "lm", se = FALSE) # linear model, no standard errors
```

> **Exercise:** repeat the plot of the median, but grouping the data by `gender` as well as `birthyear`. Set the aesthetic mapping `colour` to plot the results by gender.

Histograms, density and boxplots

How many trips did each bike take? We can plot a histogram showing the number of trips per bike:

```
ggplot(data = divvy, aes(x = bikeid)) + geom_histogram(binwidth = 50)
```

showing a quite uniform density. Speaking of which, we can draw a density plot:

```
ggplot(data = divvy, aes(x = bikeid)) + geom_density()
```

Similarly, we can produce boxplots, for example showing the tripduration for men and women (in `log10`, as the distribution is close to a lognormal):

```
ggplot(data = divvy, aes(x = gender, y = log10(tripduration))) + geom_boxplot()
```

It is very easy to change geometry, for example switching to a violin plot:

```
ggplot(data = divvy, aes(x = gender, y = log10(tripduration))) + geom_violin()
```

Duration by weekday

Now we're going to test whether the trip duration varies considerably by weekday. To do so, we load the package `lubridate`, which contains many excellent functions for manipulating dates and times.

```
library(lubridate)
```

we then create a new variable, `tripday` specifying the day of the week when the trip was initiated. First, we want to transform the string `starttime` into a date:

```
head(divvy) %>% mutate(tripday = mdy_hm(starttime)) #mdy_hm specifies the date format
```

then we can call `wday` with `label = TRUE` to have a label specifying the day of the week:

```
head(divvy) %>% mutate(tripday = wday(mdy_hm(starttime), label = TRUE))
```

Looks good! Let's perform this operation on the whole set:

```
divvy <- divvy %>% mutate(tripday = wday(mdy_hm(starttime), label = TRUE))
```

Exercises:

- Produce a barplot (`geom_bar`) showing the number of trips by day
- Calculate the median trip duration per weekday. Then plot it with the command:

```
ggplot(medianbyweekday, aes(x = tripday, y = mediantrip)) +
       geom_bar(stat = "identity")
```

the command `stat = "identity"` tells `ggplot2` to interpret the `y` aesthetic mapping as the height of the barplot.

Scales

We can use scales to determine how the aesthetic mappings are displayed. For example, we could set the x axis to be in logarithmic scale, or we can choose how the colors, shapes and sizes are used. `ggplot2` uses two types of scales: `continuous` scales are used for continuos variables (e.g., real numbers); `discrete` scales for variables that can only take a certain number of values (e.g., colors, shapes, sizes).

For example, let's plot a histogram of `tripduration`:

```
ggplot(divvy, aes(x = tripduration)) + geom_histogram() # no transformation
ggplot(divvy, aes(x = tripduration)) + geom_histogram() +
  scale_x_continuous(trans = "log")
ggplot(divvy, aes(x = tripduration)) + geom_histogram() +
  scale_x_continuous(trans = "log10")
ggplot(divvy, aes(x = tripduration)) + geom_histogram() +
  scale_x_continuous(trans = "sqrt", name = "Duration in minutes")
ggplot(divvy, aes(x = tripduration)) + geom_histogram() + scale_x_log10() # shorthand
```

We can use different color scales. We can convert the capacity to a factor, to use discrete scales:

```
pl <- ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, colour = as.factor(dpcapacity)) +
  geom_point()
pl + scale_colour_brewer()
pl + scale_colour_brewer(palette = "Spectral")
pl + scale_colour_brewer(palette = "Blues")
pl + scale_colour_brewer("Station Capacity", palette = "Paired")
```

Or use the capacity as a continuous variable:

```
pl <- ggplot(data = divvy_stations) +
  aes(x = longitude, y = latitude, colour = dpcapacity) +
```

```
  geom_point()
pl + scale_colour_gradient()
pl + scale_colour_gradient(low = "red", high = "green")
pl + scale_colour_gradientn(colours = c("blue", "white", "red"))
```

Similarly, you can use scales to modify the display of the shapes of the points (`scale_shape_contintuous`, `scale_shape_discrete`), their size (`scale_size_continuous`, `scale_size_discrete`), etc. To set values manually (useful typically for discrete scales of colors or shapes), use `scale_colour_manual`, `scale_shape_manual` etc.

Themes

Themes allow you to manipulate the look and feel of a graph with just one command. The package `ggthemes` extends the themes collection of `ggplot2` considerably. For example:

```
library(ggthemes)
pl <- ggplot(divvy, aes(x = tripduration)) +
  geom_histogram() +
  scale_x_continuous(trans = "log")
pl + theme_bw() # white background
pl + theme_economist() # like in the magazine "The Economist"
pl + theme_wsj() # like "The Wall Street Journal"
```

Faceting

In many cases, we would like to produce a multi-panel graph, in which each panel shows the data for a certain combination of parameters. In `ggplot` this is called *faceting*: the command `facet_grid` is used when you want to produce a grid of panels, in which all the panels in the same row (column) have axis-ranges in common; `facet_wrap` is used when the different panels do not have axis-ranges in common.

For example:

```
pl <- ggplot(data = divvy, aes(x = log10(tripduration))) + geom_histogram(binwidth = 0.1)
show(pl)
ggplot(data = divvy, aes(x = log10(tripduration), group = gender)) +
  geom_histogram(binwidth = 0.1) + facet_grid(~gender)
ggplot(data = divvy, aes(x = log10(tripduration), group = gender)) +
  geom_histogram(binwidth = 0.1) + facet_grid(gender~.)
```

Now faceting by `tripday` and `gender`

```
ggplot(data = divvy, aes(x = log10(tripduration),
                         colour = gender,
                         fill = gender,
                         group = tripday)) +
  geom_histogram(binwidth = 0.1) +
  facet_grid(tripday~gender)
```

Setting features

Often, you want to simply set a feature (e.g., the color of the points, or their shape), rather than using it to display information (i.e., mapping some aestethic). In such cases, simply declare the feature outside the `aes`:

```
pl <- ggplot(data = divvy_stations, aes(x = longitude, y = latitude))
pl + geom_point()
pl + geom_point(colour = "red")
pl + geom_point(shape = 3)
pl + geom_point(alpha = 0.5)
```

Saving graphs

You can either save graphs as done normally in R:

```
# save to pdf format
pdf("my_output.pdf", width = 6, height = 4)
print(my_plot)
dev.off()
# save to svg format
svg("my_output.svg", width = 6, height = 4)
print(my_plot)
dev.off()
```

or use the function ggsave

```
# save current graph
ggsave("my_output.pdf")
# save a graph stored in ggplot object
ggsave(plot = my_plot, filename = "my_output.svg")
```

Multiple layers

You can overlay different plots. To do so, however, they must share some of the aesthetic mappings. The simplest case is that in which you have only one dataset:

```
ggplot(data = divvy_stations, aes(x = longitude, y = latitude)) +
  geom_density2d() +
  geom_point()
```

in this case, the geom_density2d and geom_point shared the aes, and were taken from the same dataset.

Let's build a more complicated example:

```
# Capacity of stations in Michigan Avenue
data1 <- divvy_stations %>%
  filter(grepl("Michigan", as.character(name))) %>%
  select(name, dpcapacity) %>%
  rename(value = dpcapacity)
data1

# Number of trips leaving stations in Michigan Ave
data2 <- divvy %>%
  filter(grepl("Michigan", as.character(from_station_name))) %>%
  mutate(name = from_station_name) %>%
  select(name) %>%
  group_by(name) %>%
  summarise(value = n())
data2
```

Now we want to plot the capacity:

```
pl <- ggplot(data = data1, aes(x = name, y = value)) +
    geom_point() +
    scale_y_log10() +
    theme(axis.text.x=element_text(angle=90, hjust=1)) # rotate labels
```

And overlay the other data set:

```
pl + geom_point(data = data2, colour = "red")
```

which is allowed, as the two datasets have the same `aes`. Note that Divvy should increase the capacity of the station at Michigan & Oak!

Tidying up data

The best data to plot is the one in *tidy form*, meaning that a) each variable has its own column, and b) each observation has its own row. When data is not in tidy form, you can use the package `tidy` to reshape it.

For example, suppose we want to produce a table in which we have the number of trips departing a certain station by gender. First, we create a summary:

```
station_gender <- divvy %>%
  group_by(from_station_name, gender) %>%
  summarise(tot_trips = n()) %>%
  filter(gender == "Male" | gender == "Female") %>%
  ungroup()
```

Now we would like to create two columns (for `Male` and `Female`), containing the number of trips. To do so, we **spread** the column gender:

```
# Syntax:
# my_tbl %>% spread(COL_TO_SPREAD, WHAT_TO_USE_AS_VALUE, OPTIONAL: fill = NA)
station_gender <- station_gender %>% spread(gender, tot_trips)
station_gender
```

Having reshaped the data, we can see that the station with the highest proportion of women is in Hyde Park:

```
station_gender %>%
  mutate(proportion_female = Female / (Male + Female)) %>%
  arrange(desc(proportion_female))
```

In the data, we have a column from the station of departure and one for that of arrival. Suppose that for our analysis we would need only one column for the station name, and a separate column detailing whether this is the start or the end of the trip:

```
all_stations <- divvy %>% select(from_station_name, to_station_name, tripduration)
```

We can **gather** the two columns creating a column specifying whether it's a from/to station, and one containing the name of the station:

```
# Syntax:
# my_tbl %>% gather(NAME_NEW_COL, NAME_CONTENT, COLS_TO_GATHER)
all_stations %>% gather("FromTo", "StationName", 1:2)
```

Finally, sometimes we need to split the content of a column into several columns. We can use **separate** to do this quickly:

```
divvy %>% select(starttime) %>% separate(starttime, into = c("Day", "Time"), sep = " ")
```

Joining tables

If you have multiple data frames or `tbl` objects with columns in common, it is easy to join them (as in a database). To showcase this, we are going to create a map of all the trips in the data. First, we count the number of trips from/to each pair of stations:

```
num_trips <- divvy %>% group_by(from_station_id, to_station_id) %>% summarise(trips = n())
# remove trips starting and ending at the same point, for easier visualization
num_trips <- num_trips %>% filter(from_station_id != to_station_id)
```

Now we use `inner_join` to combine the data from `num_trips` and `divvy_stations`, creating the columns `x1` and `y1` containing the coordinates of the starting station. If we rename the columns so that their names match, the join is done automatically:

```
only_id_lat_long <- divvy_stations %>% select(id, latitude, longitude)

# Join the coordinates of the starting station
num_trips <- inner_join(num_trips,
                        only_id_lat_long %>%
                          rename(from_station_id = id,
                                 x1 = longitude,
                                 y1 = latitude))
# Join the coordinates of the ending station
num_trips <- inner_join(num_trips,
                        only_id_lat_long %>%
                          rename(to_station_id = id,
                                 x2 = longitude,
                                 y2 = latitude))

num_trips$trips <- as.numeric(num_trips$trips)

# Now we can plot all the trips!
ggplot(data = num_trips,
       aes(x = x1, y = y1, xend = x2, yend = y2,
           alpha = trips / max(trips)) )+
  geom_curve() + scale_alpha_identity() + theme_minimal()
```

Project: network analysis of Divvy data

Now that we have an overview of the methods available, we are going to perform some simple analysis on the data. First of all, we are going to create a matrix of station-to-station flows, where the rows are the starting stations, the columns the ending stations, and coefficients in the matrix measure the number of trips.

For this, we can use a combination of `dplyr` and `tidyr`:

```
flows <- divvy %>%
  select(from_station_id, to_station_id) %>%
  group_by(from_station_id, to_station_id) %>%
  summarise(trips = n())
# transform into a matrix
```

```
flows_mat <- flows %>% spread(to_station_id, trips, fill = 0) %>% as.matrix()
# remove the first col (use it for row name)
rownames(flows_mat) <- flows_mat[,1]
flows_mat <- flows_mat[,-1]
# see one corner of the matrix
flows_mat[1:10, 1:10]
```

Now we're going to rank stations according to their PageRank, the algorithm at the heart of Google's search engine. The idea of PageRank is to simulate a random walk on a set of web-pages: at each step, the random walker can follow a link (with a probability proportional its weight), or "teleport" to another page at random (with small probability). The walk therefore describes a Markov Chain, whose stationary distribution (Perron eigenvector) is the PageRank score for all the nodes. This value indicates how "central" and important a node in the network is.

Mathematically, we want to calculate the Perron eigenvector of the matrix:

$M' = (1 - \epsilon)M + \epsilon U$

Where M is a nonnegative matrix with columns summing to 1, and U is a matrix with all coefficients being 1. ϵ is the teleport probability.

First, we construct the matrix M, by dividing each row for the corresponding row sum, and transposing:

```
M <- t(flows_mat / rowSums(flows_mat))
```

Then, we choose a "teleport probability" (here $\epsilon = 0.01$), and build M':

```
U <- matrix(1, nrow(M), ncol(M))
epsilon <- 0.01
M_prime <- (1 - epsilon) * M + epsilon * U
```

and calculate the PageRank

```
ev <- eigen(M_prime)$vectors[,1]
# normalize ev
ev <- ev / sum(ev)
page_rank <- data.frame(station_id = as.integer(rownames(M_prime)), pagerank = Re(ev))
```

Which stations are the most "central" Divvy stations in Chicago? Let's plot them out:

```
st_pr <- inner_join(divvy_stations, page_rank, by = c("id" = "station_id"))
st_pr <- st_pr %>% mutate(lab = replace(name, pagerank < 0.0055, NA))
ggplot(st_pr,
       aes(x = longitude, y = latitude, colour = pagerank,
           size = pagerank, label = lab)) +
  geom_point() + geom_text(colour = "black",hjust=0, vjust=0)
```

Exercises in groups

The file `data/Chicago_Crimes_May2016.csv` contains a list of all the crimes reported in Chicago in May 2016. Form small groups and work on the following exercises:

- **Crime map** write a function that takes as input a crime's `Primary Type` (e.g., `ASSAULT`), and draws a map of all the occurrences. Mark a point for each occurrence using `Latitude` and `Longitude`. Set the `alpha` to something like 0.1 to show brighter colors in areas with many occurrences.
- **Crimes by community** write a function that takes as input a crime's `Primary Type`, and produces a barplot showing the number of crimes per `Community area`. The names of the community areas are

found in the file `data/Chicago_Crimes_CommunityAreas.csv`. You will need to `join` the tables before plotting.

- **Violent crimes** add a new column to the dataset specifying whether the crime is considered violent (e.g., `HOMICIDE`, `ASSAULT`, `KIDNAPPING`, `BATTERY`, `CRIM SEXUAL ASSAULT`, etc.)
- **Crimes in time** plot the number of violent crimes against time, faceting by community areas.
- **Dangerous day** which day of the week is the most dangerous?
- **Dangerous time** which time of the day is the most dangerous (divide the data by hour of the day).
- **Correlation between crimes** which crimes tend to have the same pattern? Divide the crimes by day and type, and plot the correlation between crimes using `geom_tile` and colouring the cells according to the correlation (see `cor` for a function that computes the correlation between different columns).