ipea Instituto de Pesquisa Econômica Aplicada

Título do capítulo	CHAPTER 5 **GTFS DATA MANIPULATION AND VISUALIZATION**
Autor(es)	Rafael H. M. Pereira Daniel Herszenhut
DOI	DOI: http://dx.doi.org/10.38116/9786556350653chap5

Título do livro	**Introduction to Urban Accessibility: a practical guide with R**
Organizadores(as)	Rafael H. M. Pereira Daniel Herszenhut
Volume	1
Série	-
Cidade	Rio de Janeiro
Editora	Instituto de Pesquisa Econômica Aplicada (Ipea)
Ano	2023
Edição	1a
ISBN	9786556350653
DOI	DOI: http://dx.doi.org/10.38116/9786556350653

5 GTFS DATA MANIPULATION AND VISUALIZATION

GTFS data is frequently used in various types of analyses that involve a few common elements. The AOP team has developed the `{gtfstools}` R package, which provides several functions that help tackling repetitive tasks and operations and facilitate feed manipulation and exploration.

In this chapter, we'll go through some of the most frequently used package features. To do this, we will use a sample of the SPTrans feed presented in the previous chapter, and which is included in the package installation.

5.1 Reading and manipulating GTFS files

Reading GTFS files with `{gtfstools}` is done with the `read_gtfs()` function, which receives a string with the file path. The package represents a feed as a list of `data.tables`, a high-performance version of `data.frames`. Throughout this chapter, we will refer to this list of tables as a *GTFS object*. By default, the function reads all `.txt` tables in the feed:

```
# loads the package
library(gtfstools)

# points to path of the sample gtfs data installed in {gtfstools}
path <- system.file("extdata/spo_gtfs.zip", package = "gtfstools")

# reads the gtfs
gtfs <- read_gtfs(path)

# checks the tables inside the gtfs object
names(gtfs)
```

```
[1] "agency"      "calendar"    "frequencies" "routes"      "shapes"
[6] "stop_times"  "stops"       "trips"
```

We can see that each `data.table` within the GTFS object is named according to the table it represents, without the `.txt` extension. This configuration allows us to select and manipulate each table individually. The code below, for example, lists the first 6 rows of the `trips` table:

```
head(gtfs$trips)
```

```
   route_id  service_id        trip_id  trip_headsign  direction_id  shape_id
1: CPTM L07         USD   CPTM   L07-0        JUNDIAI             0     17846
2: CPTM L07         USD   CPTM   L07-1            LUZ             1     17847
3: CPTM L08         USD   CPTM   L08-0   AMADOR BUENO             0     17848
4: CPTM L08         USD   CPTM   L08-1  JULIO PRESTES             1     17849
5: CPTM L09         USD   CPTM   L09-0         GRAJAU             0     17850
6: CPTM L09         USD   CPTM   L09-1         OSASCO             1     17851
```

Tables within a GTFS object can be easily manipulated using the {dplyr} or {data.table} packages, for example. In this book, we opted to use the {data.table} syntax. This package offers several useful features, primarily for manipulating tables with a large number of records, such as updating columns by reference, very fast row subsets and efficient data aggregation.[15] For example, we can use the code below to add 100 seconds to all the headways listed in the frequencies table and later reverse this change:

```
# saves original headways
original_headway <- gtfs$frequencies$headway_secs
head(gtfs$frequencies, 3)
```

```
      trip_id start_time end_time headway_secs
1: CPTM L07-0   04:00:00 04:59:00          720
2: CPTM L07-0   05:00:00 05:59:00          360
3: CPTM L07-0   06:00:00 06:59:00          360
```

```
# updates the headways
gtfs$frequencies[, headway_secs := headway_secs + 100]
head(gtfs$frequencies, 3)
```

```
      trip_id start_time end_time headway_secs
1: CPTM L07-0   04:00:00 04:59:00          820
2: CPTM L07-0   05:00:00 05:59:00          460
3: CPTM L07-0   06:00:00 06:59:00          460
```

```
# restores the original headway
gtfs$frequencies[, headway_secs := original_headway]
head(gtfs$frequencies, 3)
```

```
      trip_id start_time end_time headway_secs
1: CPTM L07-0   04:00:00 04:59:00          720
2: CPTM L07-0   05:00:00 05:59:00          360
3: CPTM L07-0   06:00:00 06:59:00          360
```

15. More details on {data.table} usage and syntax are available at: https://rdatatable.gitlab.io/data.table/index.html.

After editing a GTFS object in R, we often want to use the processed GTFS to perform different analyses. In order to do this, we frequently need the GTFS file in `.zip` format again, and not as a list of tables in an R session. To transform GTFS objects that exist in an R session into GTFS files saved to disk, `{gtfstools}` includes the `write_gtfs()` function. To use this function, we only need to pass the object that should be written to disk and the file path where it should be written to:

```
# points to the path where the GTFS should be written to
export_path <- tempfile("new_gtfs", fileext = ".zip")

# writes the GTFS to the path
write_gtfs(gtfs, path = export_path)

# lists files within the feed
zip::zip_list(export_path)[, c("filename", "compressed_size",
"timestamp")]
```

```
         filename compressed_size           timestamp
1      agency.txt             112 2023-06-16 15:38:14
2    calendar.txt             129 2023-06-16 15:38:14
3 frequencies.txt            2381 2023-06-16 15:38:14
4      routes.txt             659 2023-06-16 15:38:14
5      shapes.txt          160470 2023-06-16 15:38:14
6   stop_times.txt            7907 2023-06-16 15:38:14
7       stops.txt           18797 2023-06-16 15:38:14
8       trips.txt             717 2023-06-16 15:38:14
```

5.2 Calculating trip speed

GTFS files are often used in public transport routing applications and to inform the timetable of different routes in a given region to potential passengers. Feeds must, therefore, accurately describe the schedule and the operational speed of public transport trips.

To calculate the average speed of the trips described in a feed, `{gtfstools}` package includes the function `get_trip_speed()`. By default, the function returns the speed (in km/h) of all trips included in the feed, but one can choose to calculate the speed of selected trips with the `trip_id` parameter:

```
# calculates the speeds of all trips
speeds <- get_trip_speed(gtfs)

head(speeds)
```

```
     trip_id origin_file      speed
1: 2002-10-0      shapes   8.952511
2: 2105-10-0      shapes  10.253365
3: 2105-10-1      shapes   9.795292
4: 2161-10-0      shapes  11.182534
5: 2161-10-1      shapes  11.784458
6: 4491-10-0      shapes  13.203560
```

```
nrow(speeds)
```

```
[1] 36
```

```
# calculates the speeds of two specific trips
speeds <- get_trip_speed(gtfs, trip_id = c("CPTM L07-0",
"2002-10-0"))

speeds
```

```
      trip_id origin_file     speed
1:  2002-10-0      shapes  8.952511
2: CPTM L07-0      shapes 26.787768
```

To calculate the speed of a trip, we need to know its length and how long it takes to travel from its first to its last stop. Behind the scenes, `get_trip_speed()` uses two other functions from `{gtfstools}` toolset: `get_trip_length()` and `get_trip_duration()`. The usage of both is very similar to what has been shown before, returning the length/duration of all trips by default or of a few selected trips if desired. Below, we show their default behavior:

```
# calculates the length of all trips
lengths <- get_trip_length(gtfs, file = "shapes")

head(lengths)
```

```
     trip_id   length origin_file
1: CPTM L07-0 60.71894      shapes
2: CPTM L07-1 60.71894      shapes
3: CPTM L08-0 41.79037      shapes
4: CPTM L08-1 41.79037      shapes
5: CPTM L09-0 31.88906      shapes
6: CPTM L09-1 31.88906      shapes
```

```
# calculates the duration of all trips
durations <- get_trip_duration(gtfs)

head(durations)
```

```
     trip_id duration
1: 2002-10-0       48
2: 2105-10-0      108
3: 2105-10-1      111
4: 2161-10-0       94
5: 2161-10-1       93
6: 4491-10-0       69
```

Just as `get_trip_speed()` returns speeds in km/h by default, `get_trip_length()` returns lengths in km and `get_trip_duration()` returns the duration in minutes. These units can be adjusted with the `unit` parameter, present in all three functions.

5.3 Combining and filtering feeds

The tasks of processing and manipulating GTFS files are often performed manually, which may increase the chances of leaving minor inconsistencies or errors in the data. A common issue in some GTFS feeds is the presence of duplicate records in the same table. SPTrans' feed, for example, contains duplicate records both in `agency.txt` and in `calendar.txt`:

```
gtfs$agency
```

```
   agency_id agency_name                                agency_url
1:         1     SPTRANS http://www.sptrans.com.br/?versao=011019
2:         1     SPTRANS http://www.sptrans.com.br/?versao=011019
     agency_timezone agency_lang
1: America/Sao_Paulo          pt
2: America/Sao_Paulo          pt
```

```
gtfs$calendar
```

```
    service_id monday tuesday wednesday thursday friday saturday sunday
 1:        USD      1       1         1        1      1        1      1
 2:        U__      1       1         1        1      1        0      0
 3:        US_      1       1         1        1      1        1      0
 4:        _SD      0       0         0        0      0        1      1
 5:        __D      0       0         0        0      0        0      1
 6:        _S_      0       0         0        0      0        1      0
```

```
 7:       USD     1       1       1       1       1       1      1
 8:       U__     1       1       1       1       1       0      0
 9:       US_     1       1       1       1       1       1      0
10:       _SD     0       0       0       0       0       1      1
11:       __D     0       0       0       0       0       0      1
12:       _S_     0       0       0       0       0       1      0
```

```
    start_date   end_date
 1: 2008-01-01 2020-05-01
 2: 2008-01-01 2020-05-01
 3: 2008-01-01 2020-05-01
 4: 2008-01-01 2020-05-01
 5: 2008-01-01 2020-05-01
 6: 2008-01-01 2020-05-01
 7: 2008-01-01 2020-05-01
 8: 2008-01-01 2020-05-01
 9: 2008-01-01 2020-05-01
10: 2008-01-01 2020-05-01
11: 2008-01-01 2020-05-01
12: 2008-01-01 2020-05-01
```

{gtfstools} includes the `remove_duplicates()` function to keep only unique entries in all tables of the feed. This function takes a GTFS object as input and returns the same object without duplicates:

```
no_dups_gtfs <- remove_duplicates(gtfs)

no_dups_gtfs$agency
```

```
   agency_id agency_name                                agency_url
1:         1     SPTRANS http://www.sptrans.com.br/?versao=011019
     agency_timezone agency_lang
1: America/Sao_Paulo          pt
```

```
no_dups_gtfs$calendar
```

```
   service_id monday tuesday wednesday thursday friday saturday sunday
1:       USD      1       1         1        1      1        1      1
2:       U__      1       1         1        1      1        0      0
3:       US_      1       1         1        1      1        1      0
4:       _SD      0       0         0        0      0        1      1
5:       __D      0       0         0        0      0        0      1
6:       _S_      0       0         0        0      0        1      0
```

```
   start_date   end_date
1: 2008-01-01 2020-05-01
2: 2008-01-01 2020-05-01
3: 2008-01-01 2020-05-01
4: 2008-01-01 2020-05-01
5: 2008-01-01 2020-05-01
6: 2008-01-01 2020-05-01
```

We often have to deal with multiple feeds describing the same study area. For example, when the bus and the rail systems of a single city are described in separate GTFS files. In such cases, we may want to merge both files into a single feed to reduce the data processing effort. To help us with that, `{gtfstools}` includes the `merge_gtfs()` function. The example below shows the output of merging SPtrans' feed (without duplicate entries) with EPTC's feed:

```
# reads Porto Alegre's GTFS
poa_path <- system.file("extdata/poa_gtfs.zip", package =
"gtfstools")
poa_gtfs <- read_gtfs(poa_path)

poa_gtfs$agency
```

```
   agency_id
1:      EPTC
                                  agency_name
1: Empresa Publica de Transportes e Circulação
              agency_url      agency_timezone
1: http://www.eptc.com.br    America/Sao_Paulo
   agency_lang agency_phone
1:          pt          156
                                                 agency_fare_url
1: http://www2.portoalegre.rs.gov.br/eptc/default.php?p_secao=155
```

```
no_dups_gtfs$agency
```

```
   agency_id agency_name                                agency_url
1:         1     SPTRANS http://www.sptrans.com.br/?versao=011019
     agency_timezone agency_lang
1: America/Sao_Paulo          pt
```

```
# combines Porto Alegre's and São Paulo's GTFS objects
combined_gtfs <- merge_gtfs(no_dups_gtfs, poa_gtfs)

# check results
combined_gtfs$agency
```

```
   agency_id                                agency_name
1:         1                                   SPTRANS
2:      EPTC Empresa Publica de Transportes e Circulação
                                agency_url   agency_timezone agency_lang
1: http://www.sptrans.com.br/?versao=011019 America/Sao_Paulo          pt
2:                  http://www.eptc.com.br America/Sao_Paulo          pt
   agency_phone                                              agency_fare_url
1:
2:          156 http://www2.portoalegre.rs.gov.br/eptc/default.php?p_secao=155
```

We can see that the tables of both feeds are combined into a single one. This is the case when two (or more) GTFS objects contain the same table (`agency`, in the example). When a particular table is present in only one of the feeds, the function copies this table to the output. That's the case of the `frequencies` table, in our example, which exists only in SPTrans' feed:

```
names(poa_gtfs)
```

```
[1] "agency"    "calendar"    "routes"      "shapes"    "stop_times"
[6] "stops"     "trips"
```

```
names(no_dups_gtfs)
```

```
[1] "agency"      "calendar"    "frequencies" "routes"  "shapes"
[6] "stop_times"  "stops"       "trips"
```

```
names(combined_gtfs)
```

```
[1] "agency"      "calendar"    "frequencies" "routes"    "shapes"
[6] "stop_times"  "stops"       "trips"
```

```
identical(no_dups_gtfs$frequencies, combined_gtfs$frequencies)
```

```
[1] TRUE
```

Filtering feeds to keep only a few entries within each table is another operation that frequently comes up when dealing with GTFS data. Feeds are often used to describe large-scale public transport networks, which may result in complex and slow data manipulation, analysis and sharing. Thus, planners and researchers often work with feeds' subsets. If we want to measure the performance of a transport network during the morning peak, for example, we can filter our GTFS data to keep only the observations related to trips that run within this period.

{gtfstools} includes lots of functions to filter GTFS data. They are:

- filter_by_agency_id();
- filter_by_route_id();
- filter_by_service_id();
- filter_by_shape_id();
- filter_by_stop_id();
- filter_by_trip_id();
- filter_by_route_type();
- filter_by_weekday();
- filter_by_time_of_day(); and
- filter_by_sf().

5.3.1 Filtering by identifiers

The seven first functions from the above list work very similarly. They take as input a vector of identifiers and return a GTFS object whose table entries are related to the specified ids. The example below demonstrates this functionality with filter_by_trip_id():

```
# checks pre-filter object size
utils::object.size(gtfs)
```

```
864568 bytes
```

```
head(gtfs$trips[, .(trip_id, trip_headsign, shape_id)])
```

```
      trip_id trip_headsign shape_id
1: CPTM L07-0       JUNDIAI    17846
2: CPTM L07-1           LUZ    17847
3: CPTM L08-0  AMADOR BUENO    17848
4: CPTM L08-1 JULIO PRESTES    17849
5: CPTM L09-0        GRAJAU    17850
6: CPTM L09-1        OSASCO    17851
```

```
# keeps entries related to the two specified ids
filtered_gtfs <- filter_by_trip_id(
  gtfs,
  trip_id = c("CPTM L07-0", "CPTM L07-1")
)

# checks post-filter object size
utils::object.size(filtered_gtfs)
```

```
71592 bytes
```

```
head(filtered_gtfs$trips[, .(trip_id, trip_headsign, shape_id)])
```

```
      trip_id trip_headsign shape_id
1: CPTM L07-0       JUNDIAI    17846
2: CPTM L07-1           LUZ    17847
```

```
unique(filtered_gtfs$shapes$shape_id)
```

```
[1] "17846" "17847"
```

We can see from the code snippet above that the function not only filters trips, but all other tables containing a column that relates to `trip_id` in any way. The shapes of trips `CPTM L07-0` and `CPTM L07-1`, for example, are respectively described by `shape_ids` `17846` and `17847`. Therefore, these are the only shape identifiers kept in the filtered GTFS.

The function also supports the opposite behavior: instead of keeping the entries related to the specified identifiers, we can drop them. To do this, we need to set the `keep` argument to `FALSE`:

```
# removes entries related to two trips from the feed
filtered_gtfs <- filter_by_trip_id(
  gtfs,
  trip_id = c("CPTM L07-0", "CPTM L07-1"),
  keep = FALSE
)

head(filtered_gtfs$trips[, .(trip_id, trip_headsign, shape_id)])
```

```
      trip_id       trip_headsign shape_id
1: CPTM L08-0        AMADOR BUENO    17848
2: CPTM L08-1       JULIO PRESTES    17849
3: CPTM L09-0              GRAJAU    17850
4: CPTM L09-1              OSASCO    17851
5: CPTM L10-0 RIO GRANDE DA SERRA    17852
6: CPTM L10-1                BRÁS    17853
```

```
head(unique(filtered_gtfs$shapes$shape_id))
```

```
[1] "17848" "17849" "17850" "17851" "17852" "17853"
```

We can see that the specified trips, as well as their shapes, are not present in the filtered GTFS anymore. The same logic, demonstrated here with `filter_by_trip_id()`, applies to the functions that filter GTFS objects by `agency_id`, `route_id`, `service_id`, `shape_id`, `stop_id` and `route_type`.

5.3.2 Filtering by day of the week and time of the day

Another common operation when dealing with GTFS data is subsetting feeds to keep services that only happen during certain times of the day or days of the week. To do this, the package includes the `filter_by_weekday()` and `filter_by_time_of_day()` functions.

`filter_by_weekday()` takes as input the days of the week whose services that operate on them should be kept (or dropped). The function also includes a `combine` parameter, which defines how multi-days filters should work. When this argument receives the value `"and"`, only services that operate on every single specified day are kept. When it receives the value `"or"`, services that operate on at least one of the days are kept:

```
# keeps services that operate on both saturday AND sunday
filtered_gtfs <- filter_by_weekday(
  no_dups_gtfs,
  weekday = c("saturday", "sunday"),
  combine = "and"
)

filtered_gtfs$calendar[, c("service_id", "sunday", "saturday")]
```

```
   service_id sunday saturday
1:        USD      1        1
2:        _SD      1        1
```

```
# keeps services that operate EITHER on saturday OR on sunday
filtered_gtfs <- filter_by_weekday(
  no_dups_gtfs,
  weekday = c("sunday", "saturday"),
  combine = "or"
)

filtered_gtfs$calendar[, c("service_id", "sunday", "saturday")]
```

```
   service_id sunday saturday
1:        USD      1        1
2:        US_      0        1
3:        _SD      1        1
4:        __D      1        0
5:        _S_      0        1
```

`filter_by_time_of_day()`, on the other hand, takes the beginning and the end of a time window and keeps (or drops) the entries related to the trips that run within this window. The behavior of this function depends on whether a `frequencies` table is included in the feed or not: the `stop_times` timetable of trips listed in `frequencies` must not be filtered, because, as previously mentioned, it works as a reference that describes the time between consecutive stops, and the departure and arrival times listed there should not be considered rigorously. If a trip is not listed in `frequencies`, however, its `stop_times` entries are filtered according to the specified time window. Let's see how the function works with some examples:

```
# keeps trips that run within the 5am to 6am window
filtered_gtfs <- filter_by_time_of_day(gtfs, from = "05:00:00",
to = "06:00:00")

head(filtered_gtfs$frequencies)
```

```
      trip_id start_time end_time headway_secs
1: CPTM L07-0   05:00:00 05:59:00          360
2: CPTM L07-1   05:00:00 05:59:00          360
3: CPTM L08-0   05:00:00 05:59:00          480
4: CPTM L08-1   05:00:00 05:59:00          480
5: CPTM L09-0   05:00:00 05:59:00          480
6: CPTM L09-1   05:00:00 05:59:00          480
```

```
head(filtered_gtfs$stop_times[, c("trip_id", "departure_time",
"arrival_time")])
```

```
      trip_id departure_time arrival_time
1: CPTM L07-0       04:00:00     04:00:00
2: CPTM L07-0       04:08:00     04:08:00
3: CPTM L07-0       04:16:00     04:16:00
4: CPTM L07-0       04:24:00     04:24:00
5: CPTM L07-0       04:32:00     04:32:00
6: CPTM L07-0       04:40:00     04:40:00
```

```
# save the frequencies table and remove it from the original gtfs
frequencies <- gtfs$frequencies
gtfs$frequencies <- NULL

filtered_gtfs <- filter_by_time_of_day(gtfs, from = "05:00:00",
to = "06:00:00")
```

```
head(filtered_gtfs$stop_times[, c("trip_id", "departure_time",
"arrival_time")])
```

```
      trip_id departure_time arrival_time
1: CPTM L07-0       05:04:00     05:04:00
2: CPTM L07-0       05:12:00     05:12:00
3: CPTM L07-0       05:20:00     05:20:00
4: CPTM L07-0       05:28:00     05:28:00
5: CPTM L07-0       05:36:00     05:36:00
6: CPTM L07-0       05:44:00     05:44:00
```

Filtering the `stop_times` table can work in two different ways. One is to keep trips that cross the specified time window intact. The other is to keep only the timetable entries that take place *inside* this window (default behavior). This behavior is controlled by the `full_trips` parameter, as shown below (please pay attention to the times and stops present in each example):

```
# keeps any trips that cross the 5am to 6am window intact
filtered_gtfs <- filter_by_time_of_day(
  gtfs,
  from = "05:00:00",
  to = "06:00:00",
  full_trips = TRUE
)

head(
  filtered_gtfs$stop_times[
    ,
    c("trip_id", "departure_time", "arrival_time", "stop_sequence")
  ]
)
```

```
      trip_id departure_time arrival_time stop_sequence
1: CPTM L07-0       04:00:00     04:00:00             1
2: CPTM L07-0       04:08:00     04:08:00             2
3: CPTM L07-0       04:16:00     04:16:00             3
4: CPTM L07-0       04:24:00     04:24:00             4
5: CPTM L07-0       04:32:00     04:32:00             5
6: CPTM L07-0       04:40:00     04:40:00             6
```

```
# keeps only the timetable entries that happen inside the 5am
# to 6am window
filtered_gtfs <- filter_by_time_of_day(
```

```
  gtfs,
  from = "05:00:00",
  to = "06:00:00",
  full_trips = FALSE
)

head(
  filtered_gtfs $stop_times[
    ,
    c("trip_id", "departure_time", "arrival_time", "stop_sequence")
  ]
)
```

```
     trip_id departure_time arrival_time stop_sequence
1: CPTM L07-0       05:04:00     05:04:00             9
2: CPTM L07-0       05:12:00     05:12:00            10
3: CPTM L07-0       05:20:00     05:20:00            11
4: CPTM L07-0       05:28:00     05:28:00            12
5: CPTM L07-0       05:36:00     05:36:00            13
6: CPTM L07-0       05:44:00     05:44:00            14
```

5.3.3 Filtering using a spatial extent

Finally, `{gtfstools}` also includes a function that allows one to filter a GTFS object using a spatial polygon. `filter_by_sf()` takes an `sf/sfc` object (spatial representation created by the `{sf}` package), or its bounding box, and keeps the entries related to trips selected by their position in relation to that spatial polygon. Although this might seem complicated, this filtering process is fairly easy to grasp once we illustrate it with an example. To demonstrate this function, we are going to filter SPTrans' feed using the bounding box of shape `68962`. With the code snippet below we show the spatial distribution of unfiltered data along with the bounding box in red:

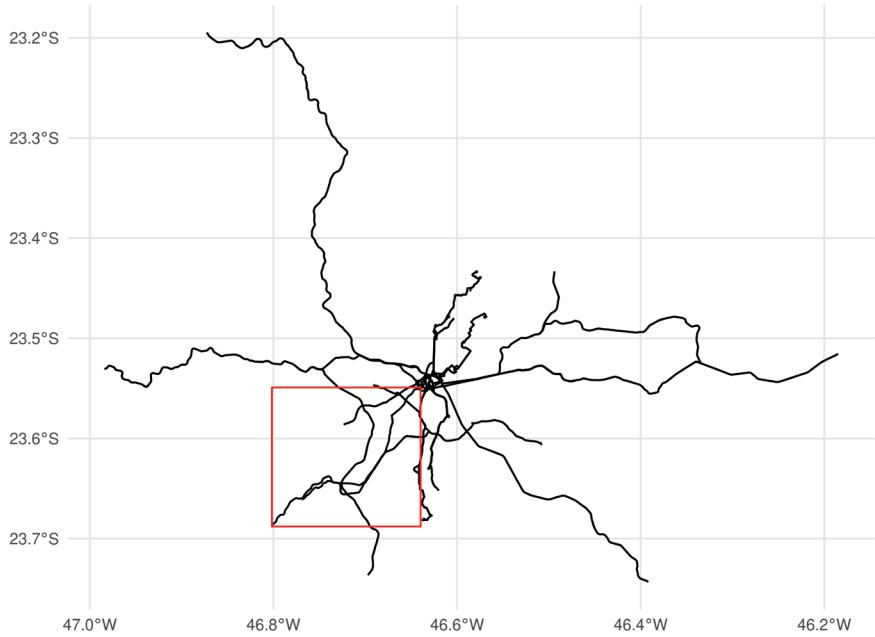
```
library(ggplot2)

# creates a polygon with the bounding box of shape 68962
shape_68962 <- convert_shapes_to_sf(gtfs, shape_id = "68962")
bbox <- sf::st_bbox(shape_68962)
bbox_geometry <- sf::st_as_sfc(bbox)

# creates a geometry with all the shapes described in the gtfs
all_shapes <- convert_shapes_to_sf(gtfs)
```

```
ggplot() +
  geom_sf(data = all_shapes) +
  geom_sf(data = bbox_geometry, fill = NA, color = "red") +
  theme_minimal()
```

FIGURE 5
Shapes spatial distribution overlayed by the bounding box of shape 68962



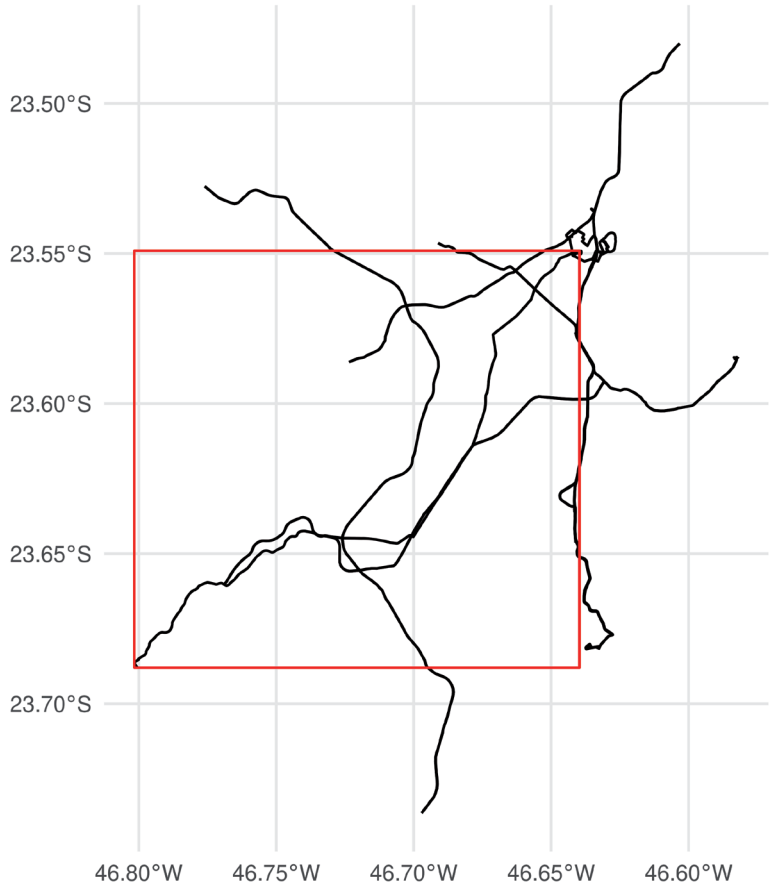
Source: Figure generated by the code snippet above.

Please note that we have used the `convert_shapes_to_sf()` function, also included in `{gtfstools}`, to convert the shapes described in the feed into a `sf` spatial object. By default, `filter_by_sf()` keeps all entries related to trips that intersect with the specified polygon:

```
filtered_gtfs <- filter_by_sf(gtfs, bbox)
filtered_shapes <- convert_shapes_to_sf(filtered_gtfs)

ggplot() +
  geom_sf(data = filtered_shapes) +
  geom_sf(data = bbox_geometry, fill = NA, color = "red") +
  theme_minimal()
```

FIGURE 6
Spatial distribution of shapes that intersect with the bounding box of shape 68962



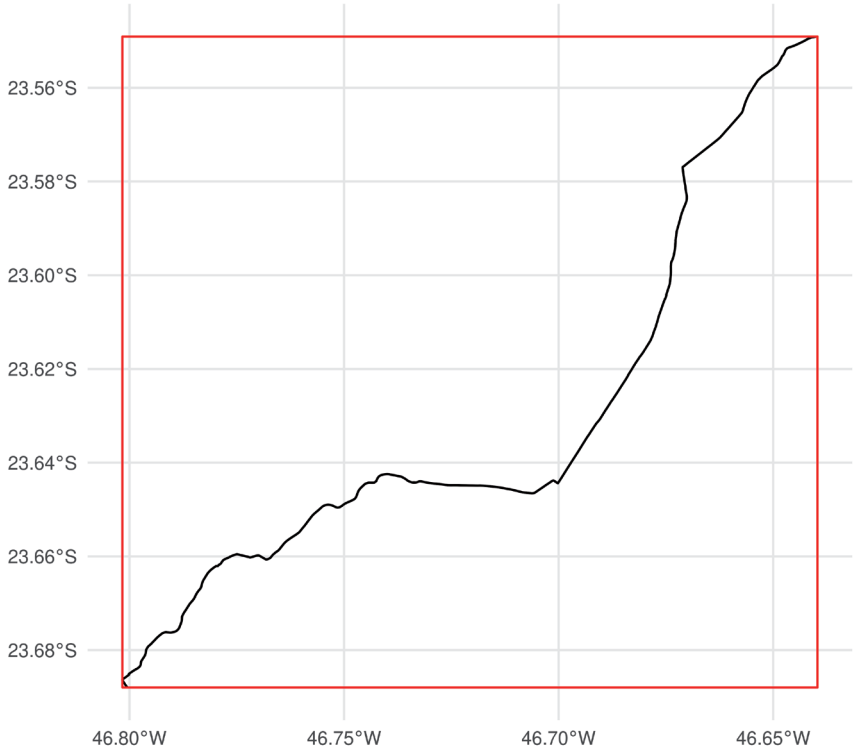
Source: Figure generated by the code snippet above.

We can, however, specify different spatial operations to filter the feed. The code below shows how we can keep the entries related to trips that are contained by the specified polygon:

```
filtered_gtfs <- filter_by_sf(gtfs, bbox, spatial_operation =
sf::st_contains)
filtered_shapes <- convert_shapes_to_sf(filtered_gtfs)

ggplot() +
  geom_sf(data = filtered_shapes) +
  geom_sf(data = bbox_geometry, fill = NA, color = "red") +
  theme_minimal()
```

FIGURE 7
Spatial distribution of shapes contained by the bounding box of shape 68962



Source: Figure generated by the code snippet above.

5.4 Validating GTFS data

Transport planners and researchers often want to assess the quality of the GTFS data they are producing or using in their analyses. Are feeds structured following the best practices adopted by the larger GTFS community? Are tables and columns adequately formatted? Is the information described by the feed reasonable (trip speeds, stop locations etc.)? These are some of the questions that may arise when dealing with GTFS data.

To answer these and other questions, `{gtfstools}` includes the `validate_gtfs()` function. This function works as a wrapper to MobilityData's Canonical GTFS Validator, which requires Java version 11 or higher to run.[16]

16. For more information on how to check the installed version of Java in your computer and on how to install the required version, please check chapter 3.

Using `validate_gtfs()` is very simple. First, we need to download the validator. To do this, we use the `download_validator()` function, included in the package, which receives the path to the directory where the validator should be saved to and the version of the validator that should be downloaded (defaults to the latest available). The function returns the path to the downloaded validator:

```
tmpdir <- tempdir()

validator_path <- download_validator(tmpdir)
validator_path
```

```
[1] "/tmp/Rtmpf3LrBZ/gtfs-validator-v4.0.0.jar"
```

The second (and final) step consists in actually validating the GTFS data with `validate_gtfs()`. This function supports GTFS data in different formats: i) as a GTFS object in an R session; ii) as a path to a local GTFS file in `.zip` format; iii) as an URL pointing to a feed; or iv) as a directory containing unzipped GTFS tables. The function also takes a path to a directory where the validation result should be saved to and the path to the validator that should be used in the process. In the example below, we validate SPTrans' feed from its path:

```
output_dir <- tempfile("gtfs_validation")

validate_gtfs(
  path,
  output_path = output_dir,
  validator_path = validator_path
)

list.files(output_dir)
```

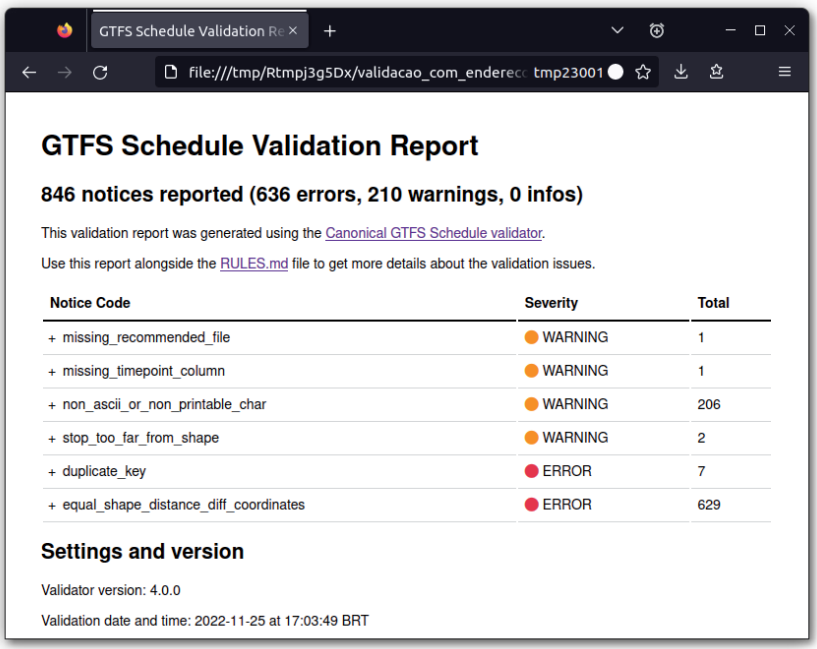
```
[1] "report.html"           "report.json"   "system_errors.json"
[4] "validation_stderr.txt"
```

We can see that the validation process generates a few output files:

- `report.html`, shown in figure 8, which summarizes the validation result in a nicely formatted HTML page (only available with validator version 3.1.0 or higher);
- `report.json`, which summarizes the same information, but in JSON format, which can be used to programatically parse and process the results;

- `system_errors.json`, which summarizes eventual system errors that may have happened during the validation process and may compromise the results; and
- `validation_stderr.txt`, which lists informative messages sent by the validator tool, including a list of the tests conducted, eventual error messages etc.[17]

FIGURE 8
Validation report example



Authors' elaboration.
Obs.: Figure whose layout and texts could not be formatted due to the technical characteristics of the original files (Publiser's note).

5.5 `{gtfstools}` workflow example: spatial visualization of headways

We have shown in previous sections that `{gtfstools}` offers a large toolset to process and analyze GTFS files. The package, however, also includes many other functions that could not be shown in this book due to space constraints.[18]

17. Informative messages may also be listed in the `validation_stdout.txt` file. Whether messages are listed in this file or in `validation_stderr.txt` depends on the validator version.
18. The complete list of functions available in `{gtfstools}` can be checked at: https://ipeagit.github.io/gtfstools/reference/index.html.

In this final section of the chapter, we illustrate how to use the package to make more complex analyses. To do this, we present a workflow that combines various functions of `{gtfstools}` together to answer the following question: how are the times between vehicles operating the same route (the headways) spatially distributed in SPTrans' GTFS?

First, we need to define the scope of our analysis. In this example, we are only going to consider the services operating during the morning peak, between 7 am and 9 am, on a typical tuesday. Thus, we need to filter our feed:

```
gtfs <- read_gtfs(path)

# filters the GTFS
filtered_gtfs <- gtfs |>
  remove_duplicates() |>
  filter_by_weekday("tuesday") |>
  filter_by_time_of_day(from = "07:00:00", to = "09:00:00")

# checking the result
filtered_gtfs$frequencies[trip_id == "2105-10-0"]
```

```
   trip_id start_time end_time headway_secs
1: 2105-10-0   07:00:00 07:59:00          900
2: 2105-10-0   08:00:00 08:59:00         1200
```

```
filtered_gtfs$calendar
```

```
   service_id monday tuesday wednesday thursday friday saturday sunday
1:        USD      1       1         1        1      1        1      1
2:        U__      1       1         1        1      1        0      0
   start_date   end_date
1: 2008-01-01 2020-05-01
2: 2008-01-01 2020-05-01
```

Next, we need to calculate the headways within this time interval. This information can be found at the `frequencies` table, though there is a factor we have to pay attention to: each trip is associated to more than one headway, as shown above (one entry for the 7 am to 7:59 am interval and another for the 8 am to 8:59 am interval). To solve this, we are going to calculate the *average* headway from 7 am to 9 am.

The first few `frequencies` rows in SPTrans' feed seem to suggest that the headways are always associated to one-hour intervals, but this is neither a rule set in the official specification nor necessarily a practice adopted by other feed

producers. Thus, we have to calculate the average headways weighted by the time duration of each headway. To do this, we need to multiply each headway by the size of the time interval during which it is valid, sum these multiplication results for each trip, and then divide this amount by the total time interval (two hours, in our case). To calculate the time intervals within which the headways are valid, we first use the `convert_time_to_seconds()` function to calculate the start and end time of the time interval in seconds and then subtract the latter by the former:

```
filtered_gtfs <- convert_time_to_seconds(filtered_gtfs)

# check how the results look like for a particular trip id
filtered_gtfs$frequencies[trip_id == "2105-10-0"]
```

```
    trip_id start_time end_time headway_secs start_time_secs end_time_secs
1: 2105-10-0   07:00:00 07:59:00          900           25200         28740
2: 2105-10-0   08:00:00 08:59:00         1200           28800         32340
```

```
filtered_gtfs$frequencies[, time_interval := end_time_secs -
start_time_secs]
```

Then we calculate the average headway:

```
average_headway <- filtered_gtfs$frequencies[,
  .(average_headway = weighted.mean(x = headway_secs,
  w = time_interval)),
  by = trip_id
]

average_headway[trip_id == "2105-10-0"]
```

```
    trip_id average_headway
1: 2105-10-0            1050
```

```
head(average_headway)
```

```
     trip_id average_headway
1: CPTM L07-0             360
2: CPTM L07-1             360
3: CPTM L08-0             300
4: CPTM L08-1             300
5: CPTM L09-0             240
6: CPTM L09-1             240
```

Now we need to generate each trip geometry and join this data to the average headways. To do this, we will use the `get_trip_geometry()` function, which returns the spatial geometries of the trips in the feed. This function allows us to specify which trips we want to generate the geometries of, so we are only going to apply the procedure to the trips present in the average headways table:

```
selected_trips <- average_headway$trip_id

geometries <- get_trip_geometry(
  filtered_gtfs,
  trip_id = selected_trips,
  file = "shapes"
)

head(geometries)
```

```
Simple feature collection with 6 features and 2 fields
Geometry type: LINESTRING
Dimension:     XY
Bounding box:  xmin: -46.98404 ymin: -23.73644 xmax: -46.63535
               ymax: -23.19474
Geodetic CRS:  WGS 84
     trip_id origin_file                        geometry
1 CPTM L07-0      shapes LINESTRING (-46.63535 -23.5...
2 CPTM L07-1      shapes LINESTRING (-46.87255 -23.1...
3 CPTM L08-0      shapes LINESTRING (-46.64073 -23.5...
4 CPTM L08-1      shapes LINESTRING (-46.98404 -23.5...
5 CPTM L09-0      shapes LINESTRING (-46.77604 -23.5...
6 CPTM L09-1      shapes LINESTRING (-46.69711 -23.7...
```

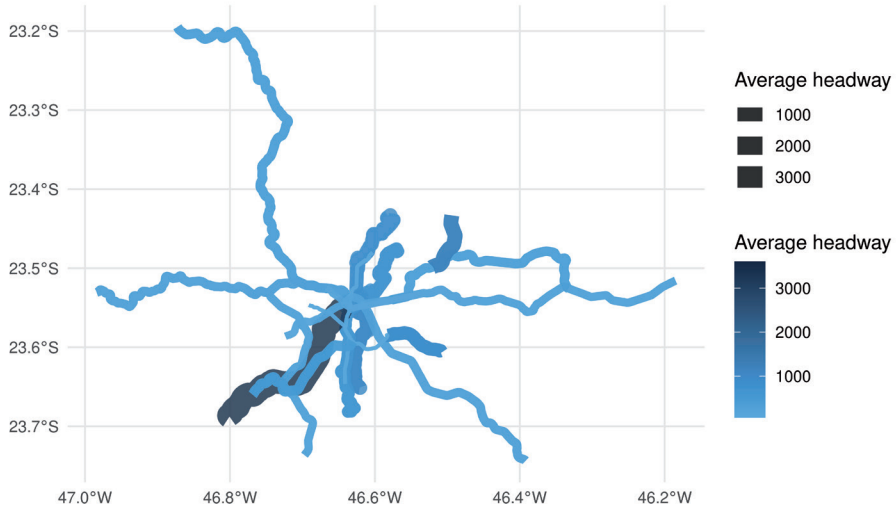
Finally, we need to join the average headway data to the geometries and then configure the map as wished. In the example below, the color and line width of each trip geometry varies with its headway:

```
geoms_with_headways <- merge(
  geometries,
  average_headway,
  by = "trip_id"
)

ggplot(geoms_with_headways) +
  geom_sf(aes(color = average_headway, size = average_headway),
  alpha = 0.8) +
```

```
    scale_color_gradient(high = "#132B43", low = "#56B1F7") +
    labs(color = "Average headway", size = "Average headway") +
    theme_minimal()
```

FIGURE 9
Headways spatial distribution in SPTrans' GTFS



Source: Figure generated by the code snippet above.

As we can see, `{gtfstools}` makes the analysis of GTFS feeds a simple task that requires only basic knowledge of table manipulation packages (such as `{data.table}` or `{dplyr}`). The example shown in this section illustrates how one could use many of the package's functions together to reveal important aspects of public transport systems specified in the GTFS format.