## Getting the data

You'll a need a Strava API account if you want to replicate this closely. However, as you'll see later in this post, you could plug in any dataframe with at least two columns: date and kilometres run at this date. As long as you have this, you won't need the Strava API. If you want to use the Strava API, you'll need to get the {rStrava} package first.

First, I'll load some packages.

```
suppressPackageStartupMessages(library(rStrava))
suppressPackageStartupMessages(library(lubridate))
suppressPackageStartupMessages(library(gganimate))
suppressPackageStartupMessages(library(scales))
```

Now, we'll get the data from the Strava API. During the authentication process, a browser window will open asking for permission to access your data.

```
app_name <- ''
app_client_id  <- ''
app_secret <- ''

# create the authentication token
stoken <- httr::config(token = strava_oauth(app_name, app_client_id, app_secret,
app_scope="activity:read_all"))

act.list <- get_activity_list(stoken)
act.df <- compile_activities(act.list)
act.df <- act.df[,c("start_date_local", "distance", "type")]
```

I've only selected the relevant variables in the data. These are `start_date_local`, the timestamp when the activity has been started, `distance`, the recorded distance in kilometres (because that is how distances are being measured!), and `type`, which could be "Run", "Ride", "Hike", "Walk" and so on. We need this to subset the activities for runs only. Note that this is not necessary - you can, of course, check for overall goals of all activites.

So, how does this dataset look like (in my case at least)?

```
scroll_box(kable_styling(kable(act.df)),
           width = "100%", height = "400px")
```

| start_date_local | distance | type |
|---|---|---|
| 2020-12-20T12:08:47Z | 6.5112 | Run |
| 2020-12-19T15:26:01Z | 3.0185 | Hike |
| 2020-12-05T17:29:16Z | 6.7965 | Run |
| 2020-11-29T14:38:19Z | 5.3272 | Hike |
| 2020-11-18T17:56:27Z | 6.8013 | Run |
| 2020-11-11T16:35:54Z | 15.1320 | Run |
| 2020-11-07T16:26:43Z | 8.0320 | Run |
| 2020-11-04T17:32:00Z | 6.8545 | Run |
| 2020-11-01T16:25:37Z | 10.6356 | Run |
| 2020-10-29T16:57:55Z | 7.0139 | Run |
| 2020-10-24T16:38:31Z | 10.0092 | Run |
| 2020-10-21T16:45:56Z | 15.0126 | Run |
| 2020-10-04T16:53:34Z | 11.1620 | Ride |
| 2020-10-04T13:44:24Z | 10.6620 | Ride |
| 2020-10-03T11:49:40Z | 8.0098 | Run |
| 2020-09-30T16:55:39Z | 15.2126 | Run |

Now **this** is exactly the point where you can plug in any dataframe - you just need the same variables. Now for…

## Preparing the data

First, we need some global variables we gonna use throughout the rest of the script. Here, you can choose other values as well.

- `goal`: (num) - That's the goal in kilometres you want(ed) to achieve. My value: 500
- `sports`: (chr) - Choose from the possible entries in the `type` column, this should also work with several types. My value: "Run"
- `selected.year`: (num) - Choose the year you want to visualise. My value: 2020
- `current.day`: (date) - This is just for the vertical line marking the current day of the year. My value: `Sys.Date()`

```
goal <- 500
sports <- "Run"
selected.year <- 2020
current.day <- Sys.Date()
```

We are selecting the relevant rows from the dataframe (year and type of sports). Then, we are sorting by timestamp and convert the timestamp into a date.

```
select.df <- act.df[act.df$type %in% sports &
                      year(act.df$start_date_local) %in% selected.year,]

select.df <- select.df[order(select.df$start_date_local),]
select.df$date <- as.Date(date(select.df$start_date_local))
```

In the next step, we are aggregating the dataset by date. Why? Because it might be possible that there are several activities at one day (e.g. when splitting up a run into two shorter ones). It leads to problems later on (when merging) if we have more than one row per date. I check before and after aggregating how many rows there are in the dataset. You will see that some aggregation has been done. By the way: Of course, we have to use `sum()` as the aggregation function because we want to get the total kilometres at each day.

```
nrow(select.df)
## [1] 52
select.df <- aggregate(distance ~ date, data = select.df, FUN = sum)
nrow(select.df)
## [1] 50
```

Now, I will introduce another dataframe, `year.df`, because our current dataset `select.df` has one problem when it comes to the visualisation later: `select.df` has one row per activity. So, unless you run *every single day of the year*, some (actually most) dates will be left out. However, for the visualisation to really work, we need a datapoint *every single day of the year*. For every day without a recorded activity, we will put zero kilometres in.

I'm sure there are other ways to do this, but I'm building `year.df` like this:

- Create `year.df` as an one-column dataframe. This column holds a sequence of dates starting at January 1st of `selected.year` and ending on December 31st of the same year. Note the `by = "day"` argument in the call to `seq()`.
- Then, I'm merging `year.df` with `select.df` by column `date`. All entries (= dates) from `year.df` should be kept. Result of this operation is a two-column `year.df` holding the additional information from `select.df`, namely kilometres per day or `NA` if no activity has been recorded at that day.
- Sort `year.df` by date
- Replace every `NA` with `0`.

```
year.df <- data.frame(date = seq(as.Date(paste0(selected.year, "-01-01")),
                                 as.Date(paste0(selected.year, "-12-31")),
                                 by = "day"))
year.df <- merge(year.df, select.df, by = "date", all.x = T, all.y = F)
year.df <- year.df[order(year.df$date),]
year.df$distance <- ifelse(is.na(year.df$distance), 0, year.df$distance)
```

Now for calculating the interesting statistics **for each day**:

- Cumulated distance (for this, the dataframe has to be sorted by date, and we did this above)
- The remaining distance till `goal`
- The remaining days until the end of the year
- The (theoretical) distance per day and week that is needed to complete the goal

We could calculate all these variables within the ggplot() call below - but this would make the plot call quite confusing. So I'm calculating them beforehand for transparency.

```
year.df$cum.distance <- round(cumsum(year.df$distance))

year.df$remaining.distance <- goal - year.df$cum.distance
year.df$days.till.end <- as.numeric(
  as.Date(paste0(selected.year, "-12-31")) -
    year.df$date)

year.df$dist.per.day.to.goal <- year.df$remaining.distance / year.df$days.till.end
year.df$dist.per.week.to.goal <- year.df$dist.per.day.to.goal * 7

year.df$dist.per.day.to.goal <- round(year.df$dist.per.day.to.goal, 1)
year.df$dist.per.week.to.goal <- round(year.df$dist.per.week.to.goal, 1)
```

Let's wrap up data preparations with some "cosmetic" stuff like setting *negative* remaining distances (when overfulfilling your goal) to 0 and inserting a dash for the final day of the year (this prevents `Inf`s from showing up in the plot).

```
year.df[year.df$dist.per.day.to.goal < 0, "dist.per.day.to.goal"] <- 0
year.df[year.df$dist.per.week.to.goal < 0, "dist.per.week.to.goal"] <- 0
year.df[year.df$remaining.distance < 0, "remaining.distance"] <- 0

year.df[year.df$days.till.end == 0, "dist.per.day.to.goal"] <- "-"
year.df[year.df$days.till.end == 0, "dist.per.week.to.goal"] <- "-"
```

Let's have another look at the resulting dataframe.

```
scroll_box(kable_styling(kable(year.df)),
           width = "100%", height = "400px")
```

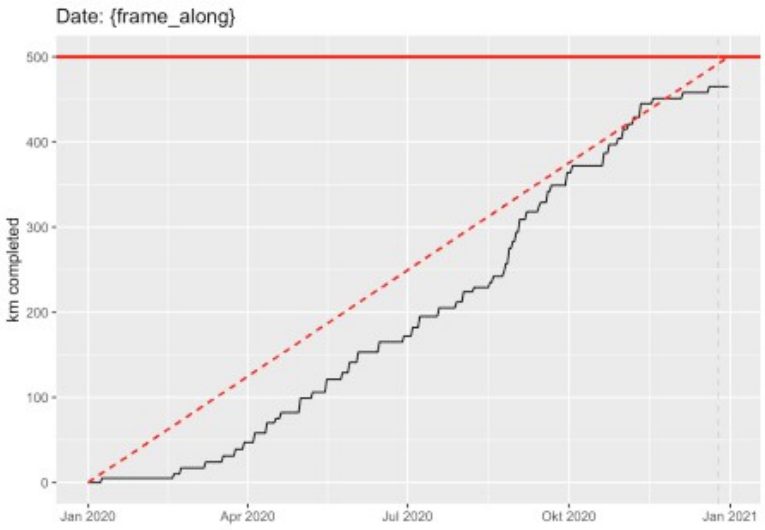
| date | distance | cum.distance | remaining.distance | days.till.end | dist.per.day.to.goal | dist.per.week.to.goal |
|---|---|---|---|---|---|---|
| 2020-01-01 | 0.0000 | 0 | 500 | 365 | 1.4 | 9.6 |
| 2020-01-02 | 0.0000 | 0 | 500 | 364 | 1.4 | 9.6 |
| 2020-01-03 | 0.0000 | 0 | 500 | 363 | 1.4 | 9.6 |
| 2020-01-04 | 0.0000 | 0 | 500 | 362 | 1.4 | 9.7 |
| 2020-01-05 | 0.0000 | 0 | 500 | 361 | 1.4 | 9.7 |
| 2020-01-06 | 0.0000 | 0 | 500 | 360 | 1.4 | 9.7 |
| 2020-01-07 | 0.0000 | 0 | 500 | 359 | 1.4 | 9.7 |
| 2020-01-08 | 0.0000 | 0 | 500 | 358 | 1.4 | 9.8 |
| 2020-01-09 | 5.0218 | 5 | 495 | 357 | 1.4 | 9.7 |
| 2020-01-10 | 0.0000 | 5 | 495 | 356 | 1.4 | 9.7 |
| 2020-01-11 | 0.0000 | 5 | 495 | 355 | 1.4 | 9.8 |
| 2020-01-12 | 0.0000 | 5 | 495 | 354 | 1.4 | 9.8 |
| 2020-01-13 | 0.0000 | 5 | 495 | 353 | 1.4 | 9.8 |
| 2020-01-14 | 0.0000 | 5 | 495 | 352 | 1.4 | 9.8 |
| 2020-01-15 | 0.0000 | 5 | 495 | 351 | 1.4 | 9.9 |
| 2020-01-16 | 0.0000 | 5 | 495 | 350 | 1.4 | 9.9 |

## Plotting and animating

I will start with the static plot and three elements:

- The line of cumulated kilometres itself (the “steps”)
- A horizontal line at 500 km (= `goal`)
- A vertical line at `current.day`
- A middle diagonal giving an impression of the “optimal” way to achieve the goal - steps under the line indicate “underperforming” and steps above the line indicate “overperforming”
- Labels (the {frame_along} placeholder will be filled by {gganimate} later)

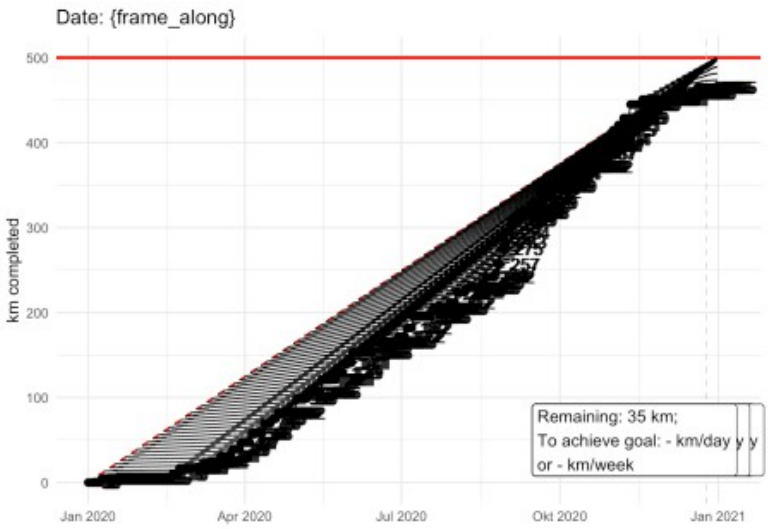
```
p <- ggplot(year.df, aes(x = date, y = cum.distance)) +
  geom_line() +
  geom_hline(yintercept = goal, col = "red", lwd = 1) +
  geom_vline(xintercept = current.day, col = alpha("grey", .5), lty = "dashed") +
  geom_segment(x = as.Date(paste0(selected.year, "-01-01")), y = 0,
                       xend = as.Date(paste0(selected.year, "-12-31")),
                       yend = goal, lty = "dashed", col = "red") +
  labs(x = "", y = "km completed", title = "Date: {frame_along}")
p
```




Now, we’re adding more elements to the plot:

- A diagonal line connecting the current point to December 31st - this shows the “optimal” way to achieve the goal from the current point in time. The steeper this line gets, the harder it is to reach the goal until the end of the year.
- A point at the end of the line
- A text label indicating the current value of cumulated kilometres at the end of the line
- A large label in the lower right part of the plot indicating the current status of the relevant statistics.

```
p <- p + geom_segment(aes(x = date, y = cum.distance),
               xend = as.Date(paste0(selected.year, "-12-31")),
               yend = goal, lty = "dotted") +
  geom_point(size = 2) +
  geom_text(aes(date + 7,
                label = round(cum.distance)),
            hjust = 0, size = 4) +
  geom_label(aes(label = paste0("Remaining: ", remaining.distance, " km;\n",
                                "To achieve goal: ", dist.per.day.to.goal, "
km/day\n",
                                "or ", dist.per.week.to.goal, " km/week"),
                 x = as.Date(paste0(selected.year, "-09-15")),
                 y = goal / 10),
             hjust = 0) +
  coord_cartesian(clip = 'off') +
  theme_minimal()
p
```




Well, this doesn't look too good, right? It's because all the animation phases (at least for the line) are plotted above each other. We need to add the animation element from {gganimate}. Here, `transition_reveal(date)` reveals the different phases along the `date` variable. Note that there are also other animation elements available for {gganimate}.

```
p <- p + transition_reveal(date)
```

And the final step: Animating the whole thing and write it into an MP4 file:

```
animate(p, fps = 60, duration = 30, width = 1440, height = 900,
        res = 200, end_pause = 25,
        renderer = av_renderer(file = "runs.mp4"))
```