g4sr vignette

Accessing Go4Schools data by API using the g4sr package.

Afsar Chowdhury*

31 August, 2021

Introduction

The g4sr package is an R wrapper for accessing school data using Go4Schools's API.

The functions you are able to use in this package are limited to the level of access you have been granted by your school's administrator.

Why this package?

Quick answer

You can access nearly all your data using just one line of code.

Example 1: I want my school's 2019 GCSE results. I type the following and hit enter:

```
gfs_clean_exam_results(academicYear = 2019, yearGroup = "11", type = "gcse")
```

Example 2: I want a list of all the students who are entitled to extra time in their assessments:

```
gfs_clean_student_send_search(academicYear = 2021, notesSearch = "extra time")
```

Example 3: I want a list of all attendances for the day:

```
gfs_clean_attendance_student_session(academicYear = 2021, goDate = "2021-07-12")
```

Example 4: I want my class list for the year:

```
gfs_clean_class_list_teacher(academicYear = 2021, staffCode = "ACH")
```

These are just a few examples.

Longer answer

If you work in a school, you often find yourself having to trawl through a million clicks of a mouse to get to the data you are after. Worse yet: you have to repeat these same clicks on a regular basis because you are an attendance officer and you want the latest attendance figures, or because you are a senior teacher and you want the latest progress in a particular subject, or because you are a teacher and the students in your class have changed, or because . . . you get the idea.

If you use Go4Schools in your school, it is possible, *in principle*, to replace all your manual clicks into these simple lines of code. The advantage of doing this is that you can run these lines of code automatically on a schedule.

*Science Department, Hyde High School, a.chowdhury@hydehighschool.uk

As someone who dabbles in data, I have a collection of functions I wrote to make my life easier. In the spirit of sharing, I have put most of these functions together in this package, hoping it will help others without their needing to know anything about coding for APIs.

Having access to the raw data in `R` also makes it possible to apply statistical models and analyses to your data far more efficiently. If you link this to your school's email system, you can send automated emails when certain parameters are met to staff, parents, or students — these could be behaviour communications, attendance communications, electronic postcards, reminders for tests, *etc.*

In this vignette, I give a brief demonstration on using the `g4sr` package.

Installation

`g4sr` can be installed from GitHub as follows:

```
# Install devtools first if needed
install.packages("devtools")

# Install g4sr using devtools
devtools::install_github("afsarchowdhury/g4sr")
```

Optional packages

There are a number of packages I have come to rely on. None of these packages are strictly necessary, but they markedly improve productivity. I recommend you install them, not least so that you are able to follow this vignette.

The first is the `tidyverse` package. This is actually a collection of packages. Second, you will also want to install the `lubridate` package to handle dates and times without going crazy.

```
# Install optional packages
install.packages("tidyverse")
install.packages("lubridate")
```

Setup

In order to use the functions in the `g4sr` package, you will need to provide a valid API key. This key should be treated like a password and never shared or disclosed to anyone else. It must never be hard-coded into a script.

Setup for beginners

If you have never used `R` before, or you only ever use it once or twice a year, the following is your best option:

```
# Load the optional packages
library(tidyverse)
library(lubridate)

# Load the g4sr package
library(g4sr)

# Run setup
gfs_setup()
```

You will be prompted to enter your API key in the console. Copy and paste your API key into the console and press enter. You will have to redo this the next time you start afresh.

Setup for more advanced users

If you are a more advanced user, you may wish to store your API key so that you don't have to re-enter it every time you start a new session. Best practice is to set the API key as an environment variable for your system and then call it in `R` using `Sys.getenv()`. If you set the parameter in `.Renviron`, it is permanently available to your `R` sessions. Be aware: if you are using version control, you do not want to commit the `.Renviron` file in your local directory. Either edit your global `.Renviron` file, or make sure that `.Renviron` is added to your `.gitignore` file.

Open the `.Renviron` file and add the following line:

```
G4SR_KEY=thisISyourAPIkeyITlooksLIKEaLONGstringOFlettersANDnumbers
```

Restart your `R` session and test to make sure the key has been added using the command:

```
Sys.getenv("G4SR_KEY")
```

You should see your API key printed in the console. If this works, you can do the following:

```
# Load the optional packages
library(tidyverse)
library(lubridate)

# Load the g4sr package
library(g4sr)
gfs_setup(api_key = Sys.getenv("G4SR_KEY"))
```

Raw and clean functions

All functions in this package start with `gfs_*`. For a complete list of all the functions that are available and how to use them, refer to the manual at https://github.com/afsarchowdhury/g4sr.

Functions that start with `gfs_clean_*` are opinionated functions that make several API calls and return data processed in a way that is useful to me. They may or may not work for you or your school.

Functions that do not have the `clean` label are raw functions where the data returned is as served by Go4Schools.

First run: `gfs_school()`

`gfs_school()` is a raw function that returns three parameters:

- the name of your school;
- the academic years that are available for you to query;
- the current academic year.

You will need these later on. For now, to run this function, you type:

```
gfs_school()
```

Here is what my output looks like:

```
## $code
## [1] "HYDETECH"
##
## $name
## [1] "Hyde High School"
##
```

```
## $academic_years
##  [1] 2021 2020 2019 2018 2017 2016 2015 2014 2013 2012
##
## $current_academic_year
## [1] 2021
```

My school name is Hyde High School and there are datasets going back to 2012 available for me to query.

It is a good idea to store this information as a variable. This will (i) limit the number of API calls you need to make, and (ii) allow you to call any element from the list downstream.

```
# Store the results of gfs_school() as my_school
my_school <- gfs_school()
```

Example 1: behaviour events over a date range

> *You are part of the behaviour team and you want to know if negative behaviour increased as the school was winding down in the last week of summer.*

The function `gfs_clean_behaviour_events_range()` takes three arguments:

- `academicYear`: academic year as an integer; you are limited to the academic years in `gfs_school()`;
- `goDateStart`: start date as a string in the form "yyyy-mm-dd";
- `goDateEnd`: end date as a string in the form "yyyy-mm-dd".

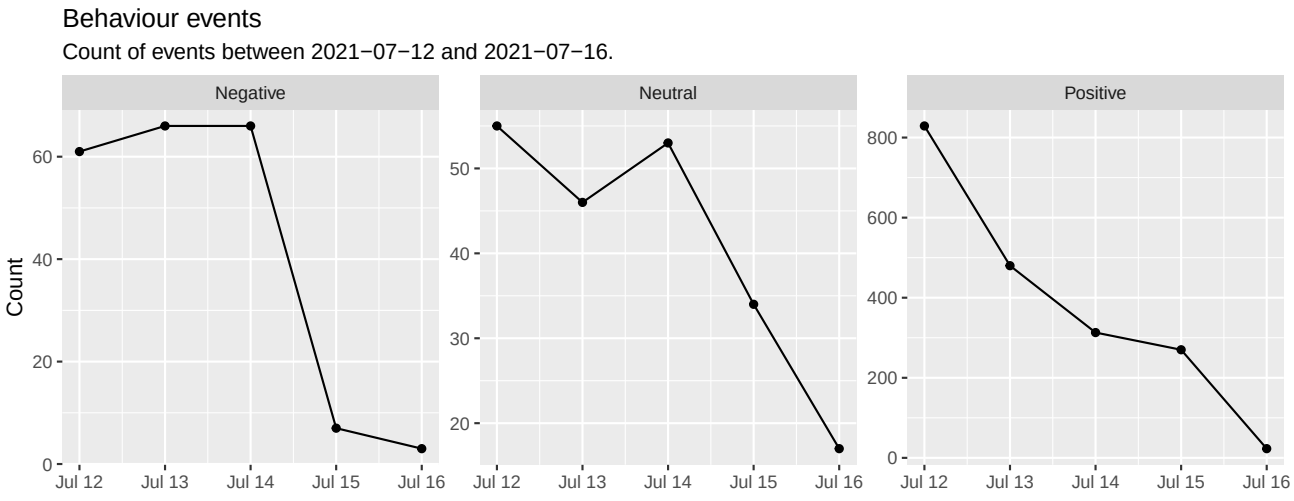
The final week at my school was 12 July, 2021, to 16 July, 2021. Therefore:

```
# Store results as df_behaviour
df_behaviour <- gfs_clean_behaviour_events_range(
  academicYear = 2021,
  goDateStart = "2021-07-12",
  goDateEnd = "2021-07-16"
)
```

The dataframe `df_behaviour` contains all recorded behaviour events between the date range given. If you need to, you can export this as a `.csv` file as follows:

```
write.csv(df_behaviour, file = "my_behaviour_data_export.csv", row.names = FALSE)
```

I cannot, for reasons to do with data protection, print my results here. I can, however, show you a quick plot of the results:



Example 2: school population trend

Resulting from a question asked in a meeting, you want to know the trend in school population.

The function `gfs_clean_school_population()` loops through your school's data and makes a note of student population in each academic year. It returns that information as a list containing the data and a line plot.

```
# Store the results as df_school
df_school <- gfs_clean_school_population()
```

To access the data from the list, type `df_school$data`. To access the plot from the list, type `df_school$plot`.

Let's have a look at the data first:

```
# View the returned data
df_school$data
```

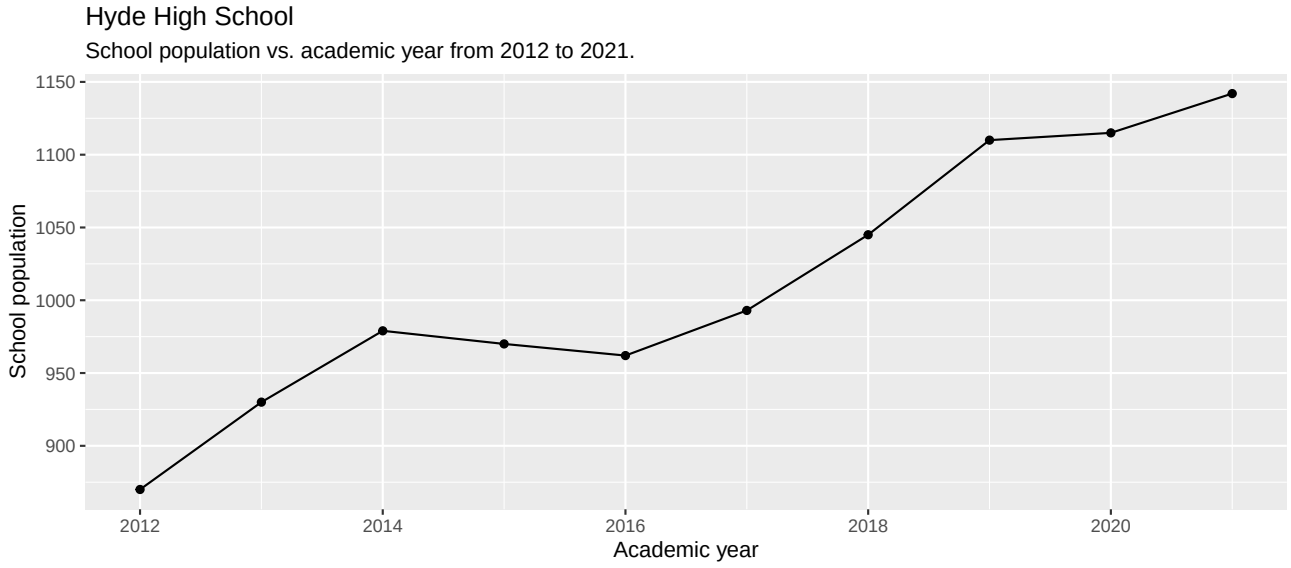
```
##    Academic.Year    n
## 1           2021 1142
## 2           2020 1115
## 3           2019 1110
## 4           2018 1045
## 5           2017  993
## 6           2016  962
## 7           2015  970
## 8           2014  979
## 9           2013  930
## 10          2012  870
```

As before, if you need to, you can export this as a `.csv` file using `R`'s built-in `write.csv()` function:

```
# Export as .csv
write.csv(df_school$data, file = "my_exported_school_data.csv", row.names = FALSE)
```

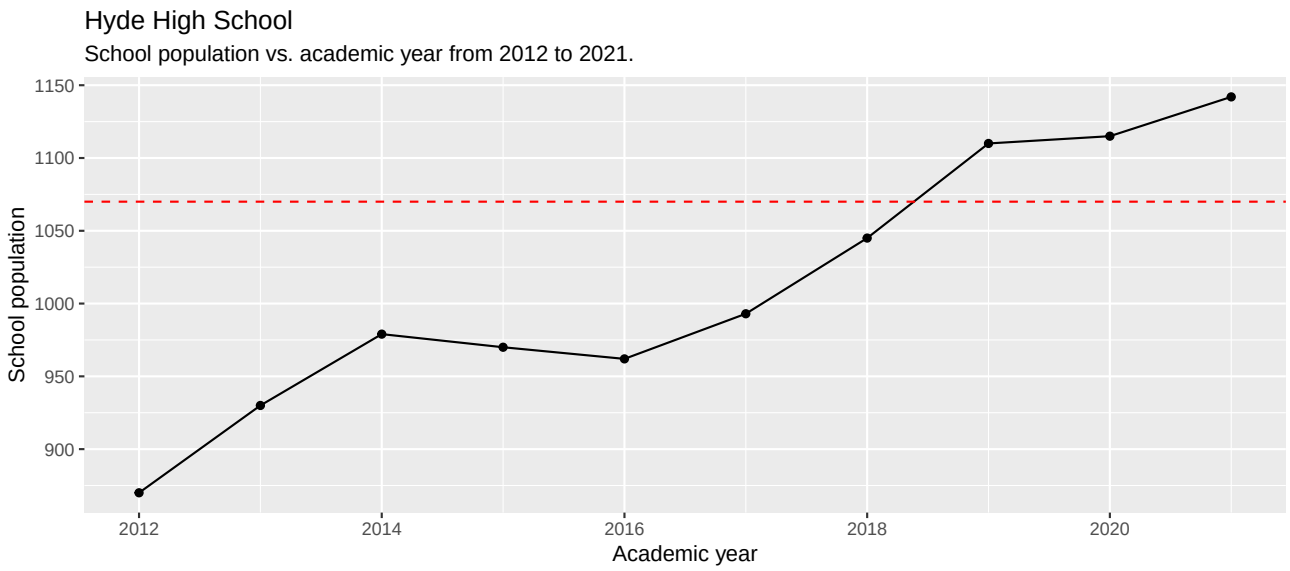
Let's have a look at the plot:

```
# View the returned plot
df_school$plot
```



The returned plot is a `ggplot2` object, which means you can apply other `ggplot2`-valid functions to the output. Hyde High School has a DfE-stipulated capacity of 1070, so let's mark this on the plot:

```
# Add a horizontal line
df_school$plot +
  geom_hline(yintercept = 1070, col = "red", lty = "dashed")
```



Example 3: lates to lesson

> *You are head of year nine and you want to know to which lessons your cohort arrives late most often. You think it would be a good idea to check for the month of June, but you don't want to include period-one lates as these are picked up by the attendance team.*

The function `gfs_clean_attendance_student_lesson_range()` returns lesson attendance details for all students over a given date range; it takes three arguments:

- `academicYear`: academic year as an integer; you are limited to the academic years in `gfs_school()`;
- `goDateStart`: start date as a string in the form "yyyy-mm-dd";
- `goDateEnd`: end date as a string in the form "yyyy-mm-dd".

```
# Store results as df_attendance_lesson
df_attendance_lesson <- gfs_clean_attendance_student_lesson_range(
  academicYear = 2021,
  goDateStart = "2021-06-01",
  goDateEnd = "2021-06-30"
)
```

You can now filter this for year nine and count the number of lates by subject and lesson:

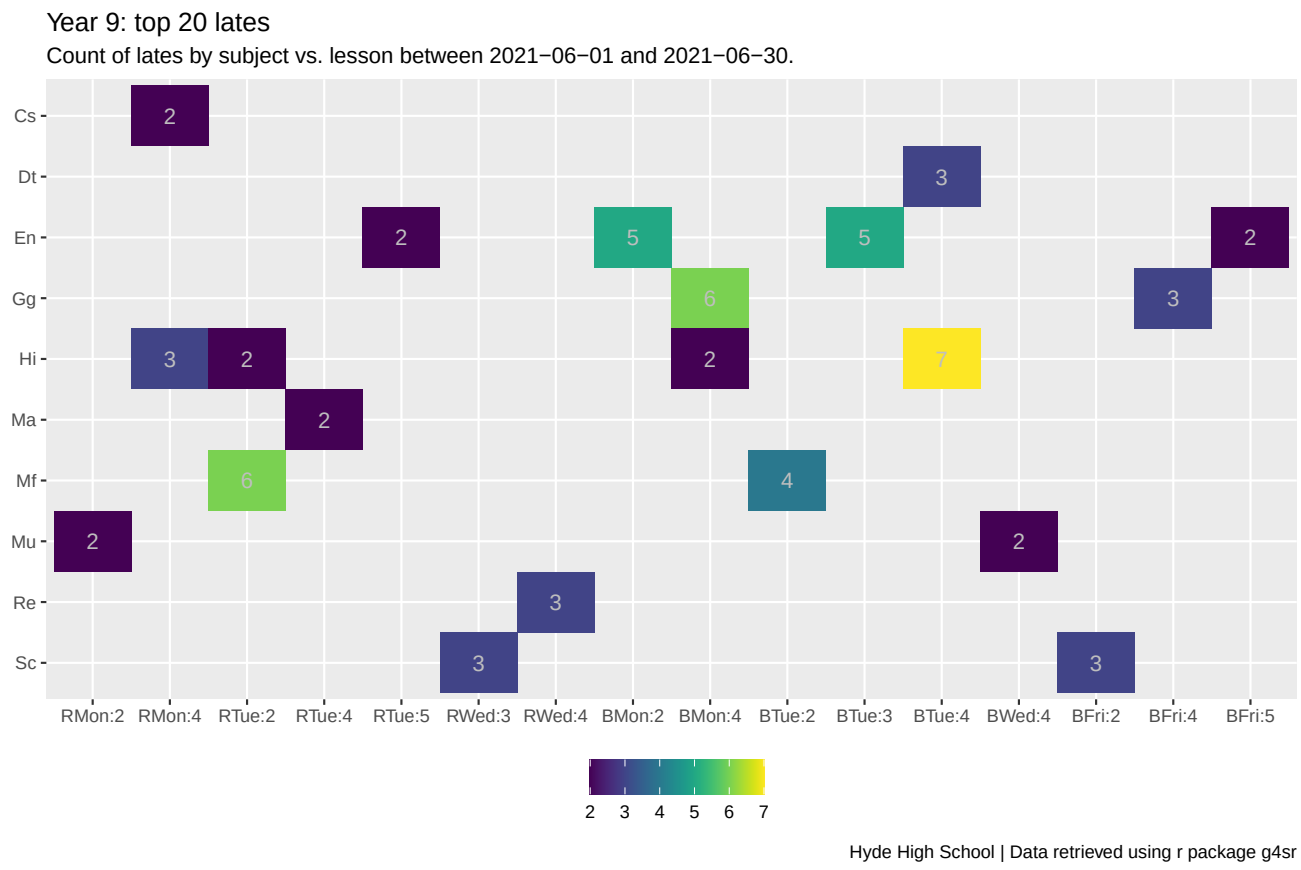
```
df_attendance_lesson %>%
  filter(!grepl(pattern = "1", x = Lesson.ID)) %>%
  filter(Year.Group == "9") %>%
  filter(Lesson.Mark == "L") %>%
  group_by(Subject, Lesson.ID) %>%
  summarise(n = n()) %>%
  ungroup() %>%
  arrange(-n)
```

Here are my top three:

```
## # A tibble: 3 x 3
```

```
##   Subject Lesson.ID     n
##   <chr>   <fct>      <int>
## 1 Hi      BTue:4         7
## 2 Gg      BMon:4         6
## 3 Mf      RTue:2         6
```

You can visualise this in the form of a tile plot:



A guided demonstration on the plotting of data is beyond the scope of this vignette. I highly recommend you read Hadley Wickham's *R for Data Science*.

Example 4: GCSE results

> *You are head of the maths department and you want to quickly export your department's GCSE results for 2019.*

The function `gfs_clean_exam_results()` returns details of all external examination results for all students in a given year group; it takes three arguments:

- `academicYear`: academic year as an integer; you are limited to the academic years in `gfs_school()`;
- `yearGroup`: year group as a string;
- `type`: type of examination as a string; for multiple types, use `type = c("gcse", "btec")`.

```
# Store the results as df_exam_results_19
df_exam_results_19 <- gfs_clean_exam_results(
  academicYear = 2019,
  yearGroup = "11",
  type = "gcse"
)
```

`df_exam_results` contains the GCSE results for all departments. To filter for maths only:

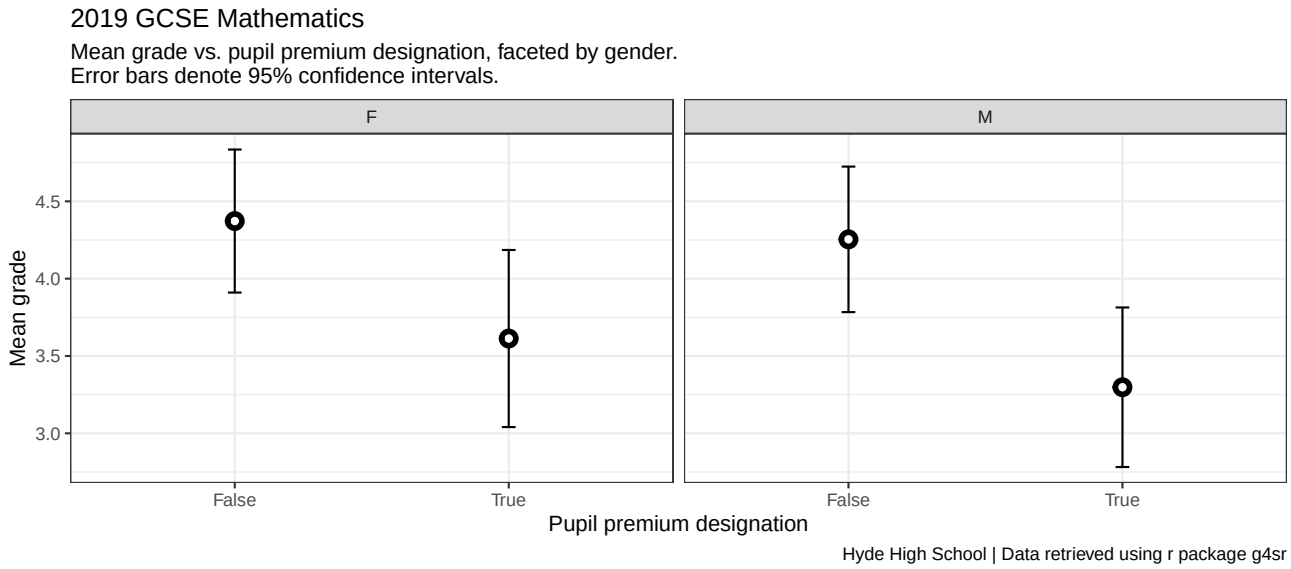
```
df_exam_results_19 <- df_exam_results %>% filter(Subject == "Mathematics")
```

To export as a `.csv` file:

```
write.csv(df_exam_results_19, file = "my_exported_results.csv", row.names = FALSE)
```

The argument `type = "gcse"` is optional and can be left out. If you do this, *all* externally assessed examination results are returned.

The returned grades are linked to staff, class, and various student demographics should you want to conduct further statistical analyses. For reasons to do with data protection, I cannot print my results here. I can, however, provide a plot of attainment by pupil premium designation and gender:



List of all clean functions

Currently, there are 14 clean functions. If these functions do not return data in the form you require, you can access the raw data from Go4Schools using the raw functions.

- `gfs_clean_attainment()`
- `gfs_clean_attainment_multiple()`
- `gfs_clean_attendance_student_lesson()`
- `gfs_clean_attendance_student_lesson_range()`
- `gfs_clean_attendance_student_session()`
- `gfs_clean_attendance_student_session_range()`
- `gfs_clean_attendance_student_summary()`
- `gfs_clean_behaviour_events_range()`
- `gfs_clean_class_list_student()`
- `gfs_clean_class_list_teacher()`
- `gfs_clean_exam_results()`
- `gfs_clean_school_population()`
- `gfs_clean_student_details_general()`
- `gfs_clean_student_send_search()`

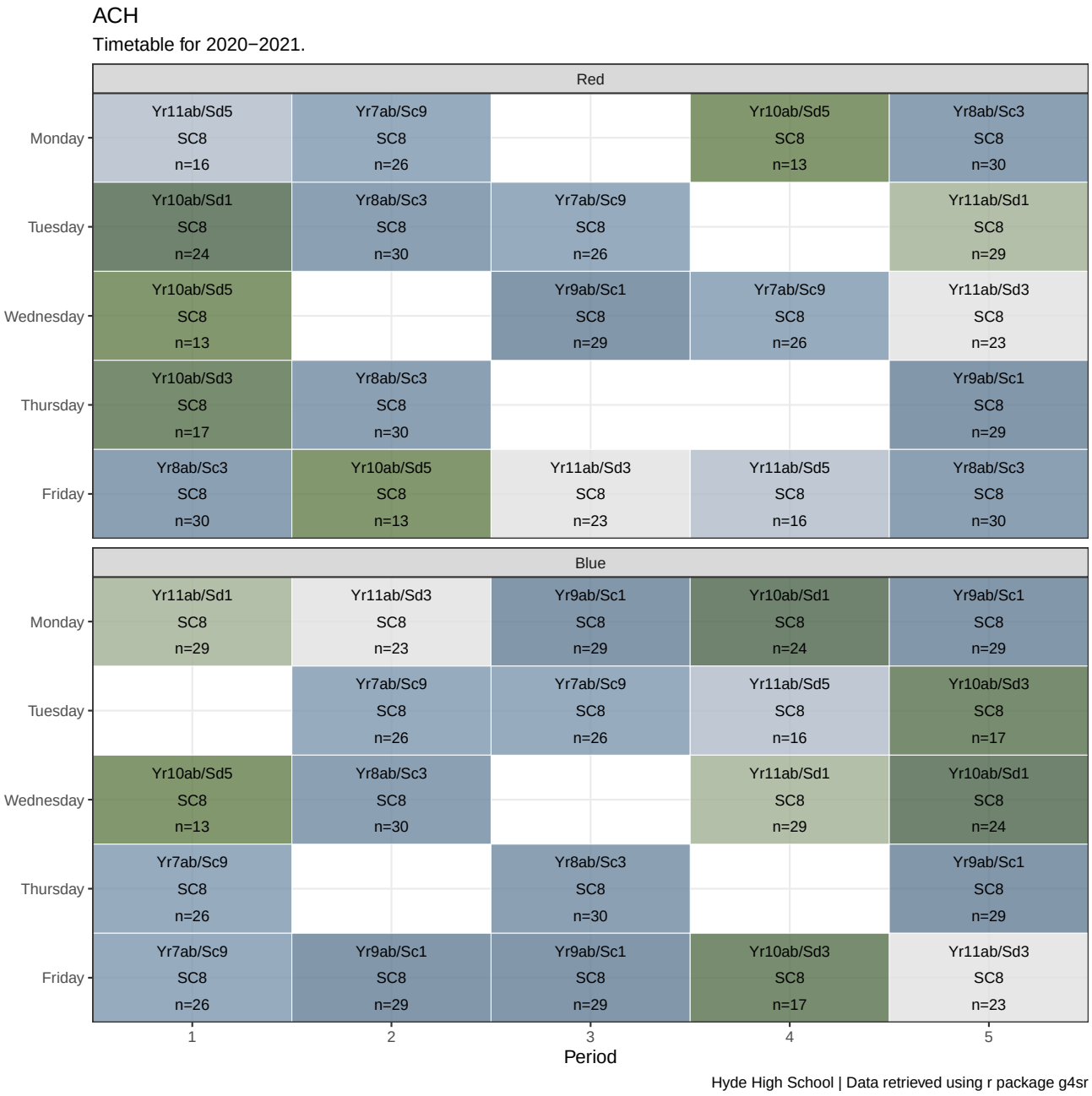
For examples on how to use all the functions in this package — raw and clean — see the manual over at https://github.com/afsarchowdhury/g4sr.

Other examples

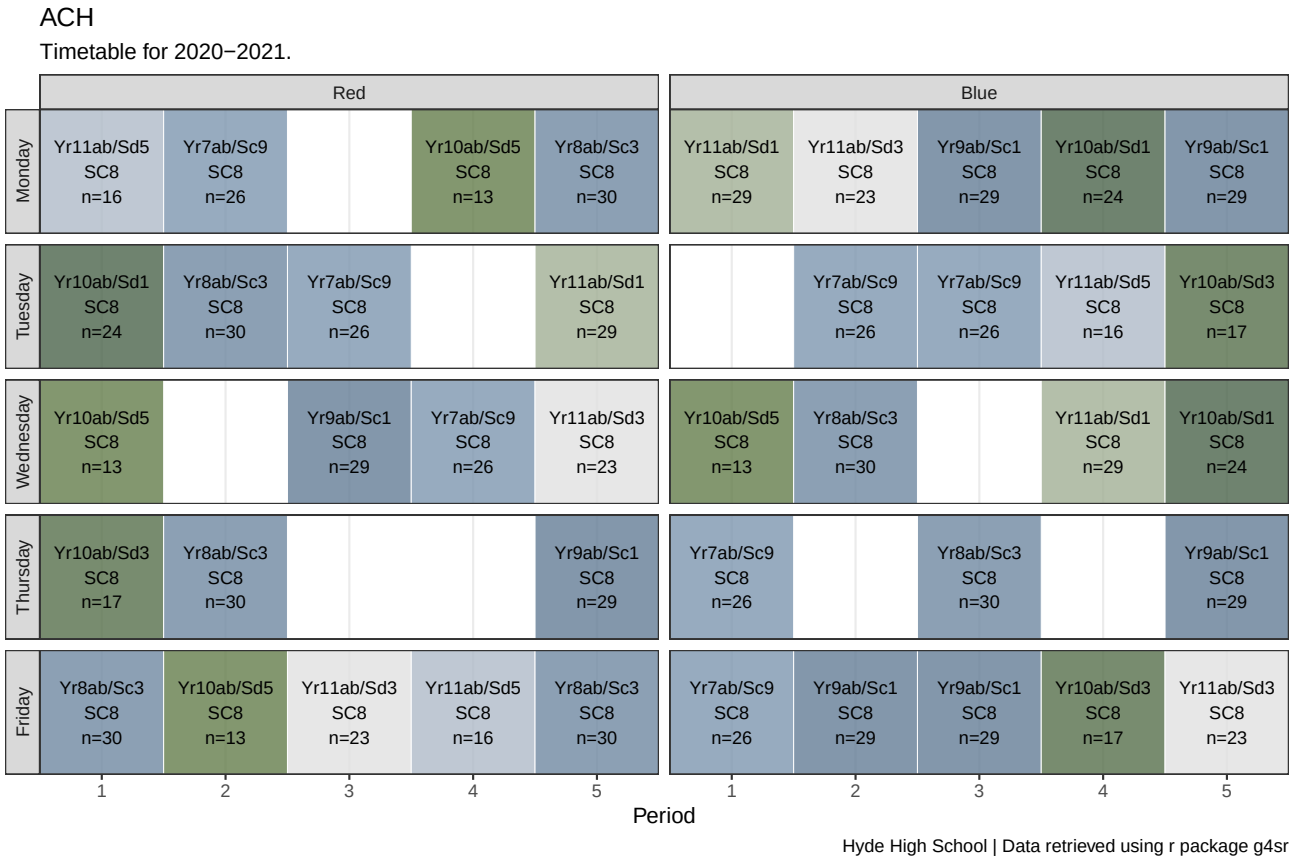
Timetables

Timetables can be generated for both staff and students. However, I have yet to turn this into a clean function that can be generalised for all schools.

In this first example, the timetable is presented in a single column containing both weeks of a two-week timetable.



In the second example, the timetable is presented in two columns.



Both examples can be adjusted for entire department timetables, or room occupancy, *etc.*

Statistical analysis

I have deliberately kept this vignette focussed on the *accessing* of data. However, the most powerful aspect of `g4sr` is that it is an `R` package, which means the data is immediately available for statistical analysis within `R`.

Scenario: *You are a head of department and your line manager tells you that your department's mean attainment this year is 3.94. Last year it was 4.22. This is a drop of more than a quarter of a grade per student. Before you make sweeping changes to the working of the department, you want to know if the difference is statistically significant.*

Applying inferential statistics to a population is a grey area in statistics. However, given that (i) some students do not turn up for exams, (ii) students often produce a range of scores if given the same test multiple times, and (ii) examiners are not error-free in their marking, treating the the two datasets as samples is, in my view, acceptable.

In this scenario there are two cohorts that are independent of one another; we are trying to out find if the mean attainment of the two cohorts is significantly different. In statistics-speak, our null hypothesis is that there is no difference:

$$H_0 : \mu_1 = \mu_2$$

where μ_1 is this year's mean attainment and μ_2 is last year's mean attainment.

One inferential test suited to this task is the two-sample t-test.

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\left[\frac{(n_1-1)\tilde{S}_1^2+(n_2-1)\tilde{S}_2^2}{n_1+n_2-2}\right]\left[\frac{1}{n_1}+\frac{1}{n_2}\right]}}$$

Exactly what these variables represent is academic, given that the point of using R is to get R to do the calculating. In broad strokes, the numerator of the equation is the difference in the mean attainment of each cohort, while the denominator is the standard error of the difference.

Alongside the metric t, the degrees of freedom needs to be calculated. These are then evaluated with a look-up table containing various confidence intervals.

$$df = n_1 + n_2 - 2$$

The entire process can be done in R as follows:

```
# Create a column containing the results for this year
pop1 <- gfs_clean_exam_results(
  academicYear = 2021,
  yearGroup = "11",
  type = "gcse"
) %>%
  filter(Subject == "your.subject") %>%
  select(-c(Staff.Code,)) %>%
  distinct() %>%
  select(c(Grade)) %>%
  mutate(Grade = as.integer(Grade))

# Create a column containing the results for last year
pop2 <- gfs_clean_exam_results(
  academicYear = 2020,
  yearGroup = "11",
  type = "gcse"
) %>%
  filter(Subject == "your.subject") %>%
  select(-c(Staff.Code,)) %>%
  distinct() %>%
  select(c(Grade)) %>%
  mutate(Grade = as.integer(Grade))

# Apply t-test
result_ttest <- t.test(pop1, pop2)
```

The object `result_ttest` contains the information you need.

Note: the following are all randomly generated data, as I cannot divulge sensitive school data that is not publicly available.

```
result_ttest
```

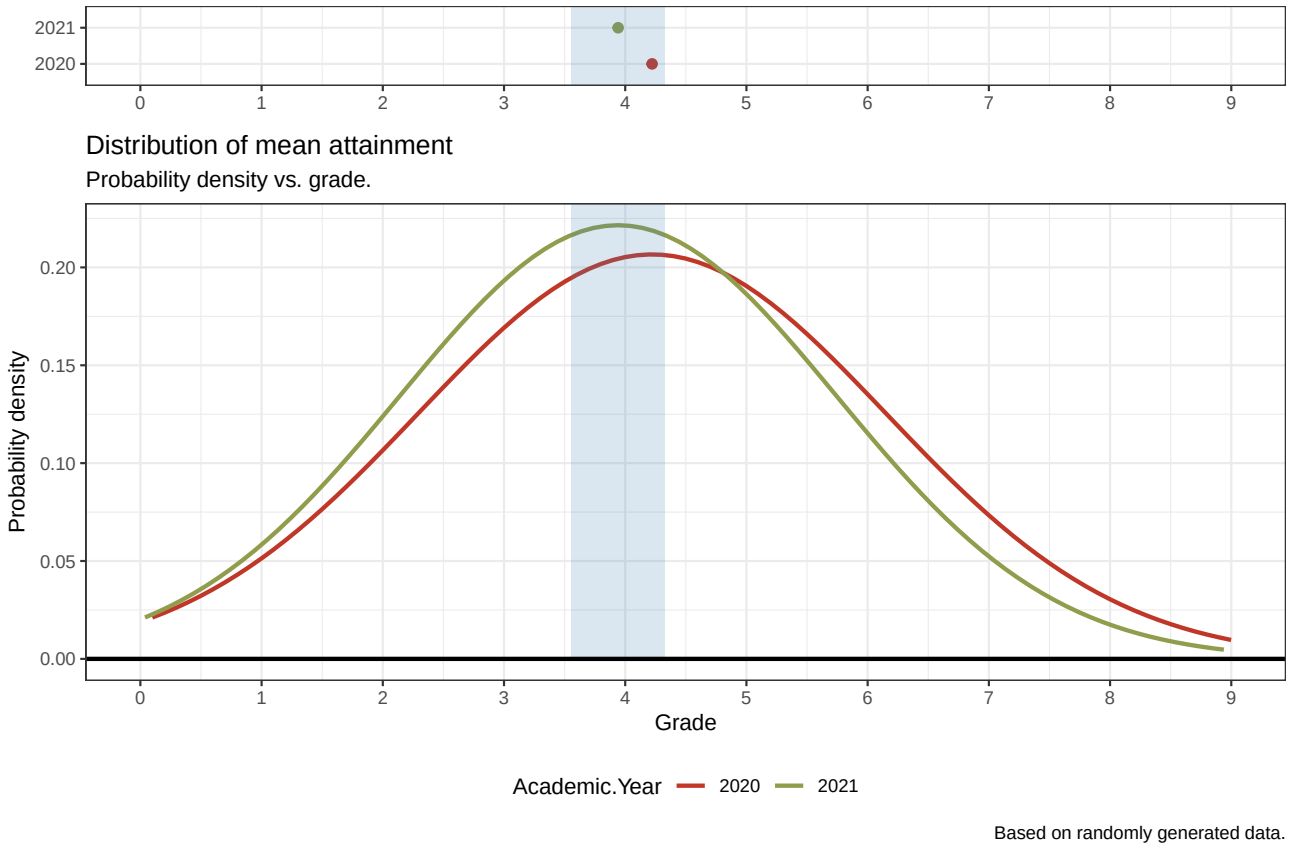
```
##
##  Welch Two Sample t-test
##
## data:  pop1 and pop2
## t = -1.4289, df = 355.47, p-value = 0.1539
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
```

```
##  -0.6656273  0.1054243
## sample estimates:
## mean of x mean of y
##  3.941489  4.221591
```

The printout shows t = -1.4288655, and the degrees of freedom = 355.4670649. If you were to look this up in a look-up table, you would find a corresponding p-value of 0.153921.

Given that $p > 0.05$, the result suggests you cannot be 95% certain that the mean attainment of the two cohorts is significantly different. You would expect a result like this to occur randomly 15% of the time.

This is a good example of where visualising the data is far more productive at demonstrating why the difference is not statistically significant. The true difference in the mean attainment of the two cohorts likely lies somewhere in the blue rectangle below.



Errors and bugs

Please log any errors and bugs over at https://github.com/afsarchowdhury/g4sr/issues.