Spotify-Streaming before and during the first wave of the COVID-19 Pandemic in Germany

Exploring and Predicting Stream Counts

Kework K. Kalustian

Contents

Introduction

The COVID-19 pandemic gives rise to several devastating consequences. In this vein, people have to isolate themselves and have to stay at home because of physical distancing orders and nationwide or, respectively, local lockdowns. Interestingly, people seem to cope with this limiting situation by considering (more) musical practices (collective singing on balconies) and by using (more) musical media, such as streaming services of different providers (e.g., Apple Music, Deezer, Spotify, YouTube, etc.). As this type of coping strategy goes in line with findings within the scopes of music cognition, music psychology, it appears particularly interesting to observe such behavior during the COVID-19 pandemic. Accordingly, it is also currently a matter of recent musicological research to explore and to underpin reasons for this kind of musical media use (see e.g., Max Planck Institute for Empirical Aesthetics, 2021). Possible reasons for this kind of music use might be that music listening and/or music making not only can be considered as a (tacit) coping strategy of music listeners to foster their well-being. Since people have to stay at home, they have potentially also more time to listen to music. Another reason for such music use might be that music streaming during this pandemic can be considered as a substitute for concerts that have been canceled due to the restrictions of the pandemic. Building on these quite general observations and assumptions, we can raise the following general question:

To what extent listen people more music during the COVID-19 pandemic compared with the period

before the pandemic at all and how far is music listening during the COVID-19 pandemic predictable as a function of date? For answering this question, we need, of course, data of recent and past music use, such as stream counts. Fortunately, the streaming service provider Spotify (2020) provides data of such stream counts so that they can be retrieved by web-scraping. Hence, we can now derive and state more concrete hypotheses from the previous question that could be interesting to test and, respectively, examine so that we can answer the research question according to these hypotheses:

- H_1: People tend to stream during the COVID-19 pandemic considerably more of Germany's daily top 200 Spotify charts than before.
- H_2: The streaming counts of Germany's daily top 200 Spotify charts during the COVID-19 pandemic are predictable with sufficient reliability as a function of date.

Since we have now formulated a general question that we aim to answer by testing these introduced hypotheses, and since we have narrowed down our interest as well, we can begin to explore the streaming count data in this vein.

1 Preparing the Working Environment, Retrieving and Cleaning the Data

First of all, to quantitatively examine such hypotheses, we will use the scripting language and statistical environment R and, respectively, RStudio. By doing so, we need to load and, respectively, install some packages, which consist of the required functions for the analyses at stake. To load and install these packages conveniently, we should write a generic function to facilitate our workflow. Especially when we are about to load multiple packages and want to ensure reproducibility, this approach appears much smoother since we do not need to repeat the same commands again and again—to simplify the workload with little effort, provides mostly a good reward.

```
# Generating a neat function to load all desired packages (x).

load_packages <- function(x){

# Object which contains the desired packages (x) as long as they are not already
# installed.

inst.packages <- x[!x %in% installed.packages()]

# Generating a for-loop.

for (i in inst.packages) {install.packages(i, dependencies = T)}

sapply(x, require, character = T)}

# Finally, using the just generated function to load and/or install the
# following desired packages.

desired.packages <- c(# Tidy coding paradigm
                      "tidyverse", "magrittr",
                      # Data import & Web-Scraping
                      "readr", "rvest",
                      # Data frames
                      "tibble", "tsibble",
                      # Data wrangling
                      "lubridate",
                      # Graphics
                      "ggExtra", "ggrepel", "scales",
                      # Machine Learning + extra Random Forest and kNN
                      # algorithms
                      "caret", "randomForest", "ranger", "FNN",
```

```
                        # Forecasting
                        "forecast", "fpp2",
                        # Time series related tests / accuracy-metrics
                        "urca", "FuzzyR",
                        # Reporting
                        "knitr", "kableExtra")

load_packages(desired.packages)
```

The first step is done. Next, we can begin with the actual work by retrieving the targeted data that we aim to analyze in the following sections.

1.2 Retrieving Spotify's daily top 200 charts in Germany: Web-Scraping

After loading and installing the desired packages, we have now to web-scrape data from the streaming service provider Spotify to retrieve the chart positions, song/track titles, and, of course, the stream counts of each song/track for the entire period between January 1st, 2019, and June 29th, 2020 (on June 29th, 2020, curfews have been lifted in Germany). That is, since the total period lasts therefore 546 days, and since each day consists of 200 chart positions, we will compose a final dataset with 109.200 rows (546 * 200) with the following lines:

```
# First, we have to store the permanent Spotify-link.

url <- "https://spotifycharts.com/regional/de/daily/"

# Here we specify the entire streaming period (i.e., the sequence of 546 days).

streaming_period <- seq(as.Date("2019/01/01"), as.Date("2020/06/29"),
                        by = "day")

# Next, we write a generic function that combines or, respectively, concatenates
# the URLs (for the entire period) by taking the permanent link from above and a
# blank argument (x) as another argument to which the URL should refer
# (i.e., our streaming_period).

gathering_urls <- function(x){paste0(url, x)}

# Using the just created function to apply it on the streaming period to finally
# get those 546 URLs.

all_urls  <- gathering_urls(streaming_period)

# Everything looks fine thus far. Hence, we create now a function that fills
# the desired column names with the information we are going to retrieve from
# those 546 URLs (i.e., chart position, song/track title, artist, stream counts,
# and dates).

spotifyR_scrapeR <- function(x) {page <- x

# Retrieving the 200 chart positions of each day.

chart_position <- page %>%
                  read_html() %>%
                  html_nodes(".chart-table-position") %>%
                  html_text()

# Retrieving the 200 song/track titles of each day.

title <- page %>%
```

```
          read_html() %>%
          html_nodes("strong") %>%
          html_text()

# Retrieving the 200 artist names of each day.

artist <- page %>%
           read_html() %>%
           html_nodes(".chart-table-track span") %>%
           html_text()

# Retrieving the 200 stream counts of each day.

stream_count <- page %>%
                read_html() %>%
                html_nodes("td.chart-table-streams") %>%
                html_text()

# Retrieving the dates of for each day of the period.

date <- page %>%
        read_html() %>%
        html_nodes(".responsive-select~ .responsive-select+
                    .responsive-select .responsive-select-value") %>%
        html_text()

# Putting these chunks together in a table of the class.

tab <- data.frame(chart_position, title, artist, stream_count, date)

return(tab)}

# As the amount of data that should be retrieved is not that small, we can
# expect that this process will take some time. To know how long this process
# will last, we calculate the difference between the process initialization and
# its end.

init_time <- Sys.time()

# The actual process of web scraping: Applying the spotifyR_scrapeR-function
# to the object of that definitive URLs for each list element. That is, the just
# created spotifyR_scrapeR-function retrieves from each URL the desired
# information.

spotifyR <- map_df(definitive_urls, spotifyR_scrapeR)

# End time of the retrieving-process.

end_time <- Sys.time()

# Difference (i.e., processing time to retrieve the desired information).

process_time <- end_time - init_time
print(process_time)

# Exporting and saving the retrieved datatable as .csv-file.
write_csv(spotifyR, "spotifyR_charts.csv")
```

Once this time-consuming process of actual web scraping (almost one hour) is done and the dataset is exported, we can load the exported and saved dataset with this simple line of code so we do not need to rerun the web-scraping procedure when we rerun the analyses:

```
# Assigning the data to an object (bit.ly-link of my GitHub-repo)

spotifyR <- read_csv("https://bit.ly/3kNXCuC")
```

As we have now assigned our whole data to the object `spotifyR`, we should check the columns and rows of this object—at least the first few ones.

Table 1: Raw web-scraped data set

chart_position	title	artist	stream_count	date
1	Benzema	by Capital Bra	439742	01/01/2019
2	HaifischNikez Allstars	by LX	373845	01/01/2019
3	Sweet but Psycho	by Ava Max	353209	01/01/2019

At a glance at the first three lines of the loaded data, we see that we need to clean them up to get a tidy version of them. For example, the preposition "by" in the rows of the "Artist" column appears dispensable for our purposes. Hence, as such information is essentially not useful for our purposes, we can omit it without further concerns. On the other hand, some further information would be helpful, such as weekday names or song/track IDs as well as the appropriate class assignments (i.e., the `stream_count` column consists, for example, in fact numbers and not doubles, likewise the `date` column consists dates and not characters). That is, we need to clean and wrangle our dataset in this regard.

1.1 Data Cleaning

For cleaning these data, we will finally add, remove, and manipulate already given information. After doing so, we can handle the data conveniently according to our needs. The following code will do the job:

```
spotifyRR <- spotifyR %>%

# Group-wise count of distinct songs/tracks (How many times occurs
# a distinct song/track during the whole period? A maximum of 546 times
# is possible as the period lasts 546 days).

add_count(title) %>%
mutate(chart_position = as.integer(chart_position),

# gsub: Replacing of a matching string pattern by a
# replacement string (i.e., we simply omit the string "by"
# and the whitespace before the artist names).

artist = sub("by\\s", "", artist),

# Adding songIDs is useful since there could be songs/tracks
# with the same title but from different artists.

songID = group_indices(., title, artist),

# Converting/coercing stream_count as integer-class

stream_count = as.integer(stream_count),

date = as.Date(date, "%m/%d/%Y"),
```

```
# lubridate::wday: Setting days and specifying the week
# start on Monday based on ISO conventions (i.e., week
# starts on Monday = 1)

weekday = wday(date, label = T, week_start = 1))
```

Now, when taking a look at the same three lines of the cleaned dataset, we see indeed a much tidier version of it and every column is converted into the right class (characters are characters, natural numbers are integers and not doubles, and finally dates are dates). So this step is done, too.

Table 2: Cleaned and wrangled data set

chart_position	title	artist	stream_count	date	n	songID	weekday
1	Benzema	Capital Bra	439742	2019-01-01	340	314	Tue
2	HaifischNikez Allstars	LX	373845	2019-01-01	143	940	Tue
3	Sweet but Psycho	Ava Max	353209	2019-01-01	502	2297	Tue

Since the data is now cleaned up, we can finally begin to test and examine our research hypotheses in the following sections.

2 Exploring Stream Counts

To begin the analyses, it is helpful to take a look at the distribution of the raw (i.e., non-summarized) overall stream counts so that we know with what kind of data properties we have to deal with in the following analyses. From a theoretical perspective it seems more likely that a high frequency of lower stream counts and a lower frequency of higher stream counts might be observable than a symmetric ("normal") distribution of those stream counts.

```
Streams <- spotifyRR %>%
           filter(date <= as.Date("2020-03-10"))

Streams_Pan <- spotifyRR %>%
               filter(date > as.Date("2020-03-10"))

cols <- c(# Colorblind friendly blue
          "Pre-Pandemic (01.01.19-10.03.20)" = "#045a8d",
          # Colorblind friendly red
          "Pandemic (11.03.20-29.06.20)" = "#bd0026")

# Calculating the binwidth following the formula of Freedman & Diaconis (1981)
# for the upcoming histogram (see Hyndman 1995)

binw <- 2 * IQR(Streams$stream_count) / nrow(Streams)^(1/3)

binw_Pan <- 2 * IQR(Streams_Pan$stream_count) / nrow(Streams_Pan)^(1/3)

# Plotting the histograms in question.

ggplot() +

# Frequency Histogram
geom_histogram(data = Streams, aes(stream_count,
                                   color = "Pre-Pandemic (01.01.19-10.03.20)"),
               binwidth  = binw, alpha = 0.5) +

geom_histogram(data = Streams_Pan, aes(stream_count,
```

```
                                    color = "Pandemic (11.03.20-29.06.20)"),
               binwidth  = binw_Pan, alpha = 0.75) +

# Customizing the labels and break points on the x-axis
scale_x_continuous(breaks = c(min(spotifyRR$stream_count), 1e5, 2e5, 25e4, 5e5,
                              1e6, 15e5, max(spotifyRR$stream_count)),
                   labels = c("37568", "100K", "200K", "250K", "500K", "1M",
                              "1.5M", "c.1.97M"))+

# Label numbers with SI prefixes
scale_y_continuous(labels = label_number_si(accurarcy = NULL))+

# For visual purposes (i.e., avoiding that high frequencies and a heavy tail
# of high counts dominate the plot), we use here a square root-scaling of
# the axes (not of the values). That is, it is of ample importance to note
# that we do not transform the scale statistics of our data: They are still
# the same.
coord_trans(x = "sqrt", y = "sqrt")+

# Customizing the labels on both axes
labs(x = "\nStream Counts of Songs", y = "Frequency\n")+

# Legend colors and shape of the color fields
scale_color_manual(name = "Legend:", values =cols) +
scale_fill_manual(name = "Legend:", values = cols) +

guides(color = guide_legend(override.aes = list(shape = 22, size = 3))) +

# Layout
theme_bw(base_size = 14) +
theme(axis.title.y = element_text(size = 14),
      axis.title.x = element_text(size = 14),
      legend.position = "top",
      legend.key = element_rect(color = "white"),
      legend.background = element_rect(fill = "white",
                                       linetype = "solid",
                                       color = "grey90"),
      plot.margin = unit(c(.66, .33, .66, .66), "cm"))

# Saving the output (wide format)
ggsave("dist_hist.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
        width = 21.33, height = 9, dpi = 320)
```

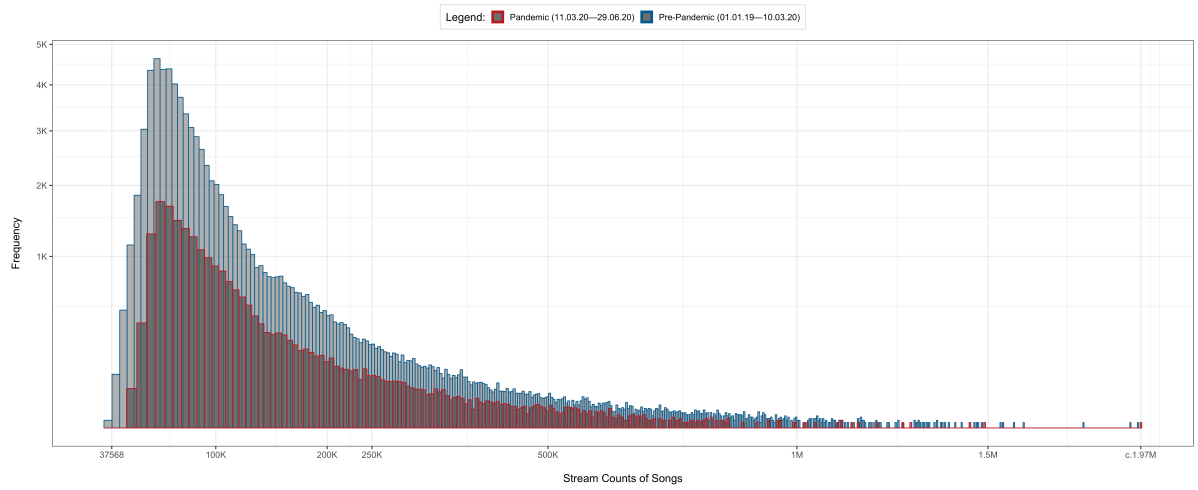

Figure 1: Histogram: Stream counts of Germany's daily Top 200 Spotfy-charts (square root-scaling).

In this combined graph we see how the frequency distributions of those stream counts are highly right-skewed or, respectively, left-leaning. That is, a higher amount of songs and tracks have been streamed between almost 40.000 and 500.000 times, whereas only few songs and tracks have been streamed more than 1.5 million times. Hence, we can already state at this stage of the descriptive analysis that the overall stream counts are not normally distributed. Based on this finding, we can now explore whether the stream counts of both periods differ in their central tendencies on a smaller level, such as summarized stream counts that are grouped by weekdays.

2.1 Differences in Weekly Stream Counts

As we now have an orientation regarding the distribution of the data, we can continue and examine whether and to what extent the stream counts of Germany's daily top 200 Spotify-charts are higher within a week interval during the COVID-19 pandemic compared to the same period in the year 2019. This endeavor refers to the first hypothesis H_1, however, under a more concrete perspective (i.e., operationalization). For testing now this hypothesis, we have first to subset, group, and summarize our data so that we can decide which test we should apply to examine potential differences. We can approach this as follows:

```
# Subsetting and summarizing the data into a pre-pandemic set

spotifyR_pre <- spotifyRR %>%
                filter(date >= as.Date("2019-03-11") &
                       date <= as.Date("2019-06-29")) %>%
                group_by(weekday) %>%
                # Using the median instead of mean because of
                # skewed overall distribution (see histogram)
                summarize(Mdn_Streams = median(stream_count))

# Subsetting and summarizing the data into a pandemic set

spotifyR_pandemic <- spotifyRR %>%
                     filter(date >= as.Date("2020-03-11") &
                            date <= as.Date("2020-06-29")) %>%
                     group_by(weekday) %>%
                     # Using the median instead of mean because of
                     # skewed overall distribution (see histogram)
                     summarize(Mdn_Streams = median(stream_count))
```

The reason for subsetting the entire dataset according to these dates relies on two facts: The first of which is that the World Health Organization (2020) has officially categorized on this date the global

Coronavirus disease as a pandemic. Secondly, on this date, the third death case in Germany has been reported according to the daily report of the Robert Koch Institute (2020). As the nationwide physical distancing orders have been valid according to German Federal Government (2020) up to June 29th, 2020, the period ends on this date for our purposes. Thus, we have good reasons to consider these dates as breaking points according to which we can distinguish two different periods.

For comparing these sets now regarding their potential difference, we should first explore the properties of this splitted dataset so that we can appropriately decide which kind of test we should perform to examine a potential difference. Since a potential difference relies on the theoretical assumption (see H_1) that a difference between these two sets regarding their central tendencies can be observed, we perform here a classic null hypothesis significance test. That is, we test in this scenario the null hypothesis according to which a significant difference is *not* observable against the alternative hypothesis according to which, in turn, exactly such a significant difference is assumed. (This is basically an indirect proof for the alternative hypothesis in that way that we could retain the alternative hypothesis, in this vein, because we had to reject the null hypothesis on the basis of the respective test result. See for an in-depth critical discussion with regard to this frequentist perspective in contrast to the Bayesian view, for instance, McElreath, 2020.)

Now, back on track, either the robust paired and exact Wilcoxon sign-rank-test or its powerful parametric analogon, the t-test, are being up to choose. As the t-test provides more test power, it also requires, however, that some assumptions, such as normality and homogeneity of variances, have to be met by the data properties, especially when it comes to small samples. But before we examine these assumptions, it might be helpful to recall very briefly the concept or rationale of the t-test.

Depending on whether differences of central tendencies of two dependent or independent sets are being up to test, different versions of the t-test need to be performed. Since we are here confronted with two depending sets (i.e., we compare a weekly period: first, during the pre-pandemic period and then during the pandemic), we need to calculate a t-test for dependent sets/samples. For doing so, first, a t-value or, respectively, a t-statistic needs to be calculated as follows (for a very accessible introduction to the t-test see also Field et al., 2012, p. 368).:

$$t = \frac{\overline{x}_d}{\sqrt{\frac{\sum_{i=1}^{n}(d_i - \overline{x}_d)^2}{(n-1)n}}} = \frac{\overline{x}_d}{\hat{\sigma}_{\overline{x}_d}}$$

Here $\overline{x}_d$ indicates the mean of the differences of the paired measured/observed values d_i (i.e, the average difference between the values from the pre-pandemic period and the values from the pandemic period). This very value ($\overline{x}_d$) is then divided by the product of the standard error of the distribution of the mean differences which is why the degrees of freedom $df = (n-1)n$ are here put in the denominator (cf., $\hat{\sigma}_{\overline{x}_d}$).

Once the t-statistic is computed, its significance (i.e., p-value) needs to be evaluated by comparing the just calculated t-value with the so-called critical value which is, on the other hand, defined by the degrees of freedom of the t-distribution in question at a pre-defined confidence or, respectively, α-level. As such a computed value, however, does not provide any information regarding the relevance of a significant result of the hypothesized difference, it appears helpful to evaluate whether such a significant difference is sufficient enough to consider it as a relevant or effective result. One prominent approach for achieving a value that provides information about the relevance in question is to use the correlation coefficient r after Pearson as an indicator for this relevance, where $r = 0$ means that the significant result of the t-test does not pose an effect at all while an r-value of 1, in turn, indicates a maximal effect (of difference). For calculating r, we need to use the t-statistic and the degrees of freedom (df) according to this formula:

$$r = \left| \sqrt{\frac{t2}{t2 + df}} \right|$$

To evaluate now whether those above-mentioned conditions are being fulfilled so that a t-test can be computed without concerns, the results of the following pre-tests have to be non-significant:

```
# Checking, whether the differences of the stream counts between the two
# periods in question are approximately normally distributed.
```

```r
# Assigning the differences to a new object. We calculate the difference
# because if the difference between those two sets is approximately normally
# distributed, so are the sets.

dif <- spotifyR_pandemic$Mdn_Streams - spotifyR_pre$Mdn_Streams

# Testing for normality of the differences
# Null hypothesis: No differences are detectable (i.e., data is normally
# distributed). That is, we "want" here a p-value greater than 0.05 at a
# confidence level of 95%.

normality <- shapiro.test(dif)

print(normality)

# Testing variance homogeneity:
# Null hypothesis: Variances of both periods are equal. That is, we "want"
# here a p-value greater than 0.05 at a confidence level of 95% (see above).

var_homo <- var.test(spotifyR_pandemic$Mdn_Streams, spotifyR_pre$Mdn_Streams)
print(var_homo)
```

These tests yield so far results that are within the desired boundaries for the t-test: The differences of the stream counts between those two periods are not significantly different from a normal distribution according to the Shapiro-Wilk-Test for normality ($W = .880, p = .228$), so we cannot reject its null hypothesis, which states exactly this property (i.e., presence of a normal distribution). Secondly, the so-called F-Test for homogeneity of variances yielded that both variances are not heterogeneous ($F_{(6,6)} = .712$, $p = .69$, $CL = 95\%$) which is why we can assume that both variances are homogeneous. Hence, we can now finally test for a potential significant difference by performing a paired one-way t-test, while recalling our hypothesis H_1, which states that people tend to stream more during the COVID-19 pandemic compared to the period before the pandemic. As we have organized our data around weekday median streaming counts, we can now consider this approach as the operationalized version of our hypothesis H_1.

```r
# Performing the t-Test

# Null hypothesis: No significant difference is detectable. That is, we "want"
# here a p-value equal to or lower than 0.05 at a confidence level of 95% since
# we hypothesized in H1 that a difference between both periods is observable in
# that way that the average of streaming counts during the pandemic is higher
# (i.e., "greater") than during the pre-pandemic period.

t <- t.test(spotifyR_pandemic$Mdn_Streams, spotifyR_pre$Mdn_Streams,
            alernative = "greater", paired = T, conf.level = .95)
print(t)

# Calculating the effect size r after Person (see second equation) by using the
# t-value (see first equation) and the degrees of freedom.

eff_size_r <- abs(sqrt(t$statistic^2 / (t$statistic^2 + t$parameter)))

print(eff_size_r)

# Central tendencies and dispersion for the (pre-)pandemic periods:

# Pre-Pandemic
mean(spotifyR_pre$Mdn_Streams); sd(spotifyR_pre$Mdn_Streams)
median(spotifyR_pre$Mdn_Streams)
```

```r
# Pandemic
mean(spotifyR_pandemic$Mdn_Streams); sd(spotifyR_pandemic$Mdn_Streams)
median(spotifyR_pandemic$Mdn_Streams)
```

So, after carrying out the one-way t-test and calculating the effect size of the observed significant result, we can state a strong and significant effect (after Cohen, 1992) of difference regarding the average central tendencies of those two sets of the weekday median stream counts and, therefore, in the music consumption of these time periods ($t_{(6)} = 6.707$, $p < .001$, $r = .939$, $CL = 95$). Resulting from this, we can now conclude that the average of the median stream counts of Germany's daily top 200 Spotify-charts within a week interval during the pandemic-period is significantly higher with a strong effect of difference compared to the pre-pandemic period ($M_{Pre-Pandemic} = 91654.14$, $SD_{Pre-Pandemic} = 9488.085$, and $Mdn_{Pre-Pandemic} = 88366$; $M_{Pandemic} = 96156.57$, $SD_{Pandemic} = 8005.661$, and $Mdn_{Pandemic} = 94015.5$). We can see this strong effect of difference in the following graph:

```r
# Creating a color-object
cols_1 <- c(# Colorblind friendly blue
          "Pre-Pandemic (11.03.19-29.06.19)" = "#045a8d",
          # Colorblind friendly red
          "Pandemic (11.03.20-29.06.20)" = "#bd0026")

ggplot(data = spotifyR_pre) +

# Connected Scatter plot
geom_point(aes(weekday, Mdn_Streams,
               color = "Pre-Pandemic (11.03.19-29.06.19)")) +

geom_line(aes(weekday, Mdn_Streams, group = 1,
              color = "Pre-Pandemic (11.03.19-29.06.19)"),
          size = .75, show.legend = F) +

geom_point(data = spotifyR_pandemic, aes(weekday, Mdn_Streams,
                                color = "Pandemic (11.03.20-29.06.20)")) +

geom_line(data = spotifyR_pandemic, aes(weekday, Mdn_Streams, group = 1,
                                color = "Pandemic (11.03.20-29.06.20)"),
          size = .75, show.legend = F) +

# Color and Legend
scale_color_manual(name = "Legend:", values = cols_1) +
scale_fill_manual(name = "Legend:", values = cols_1) +

# Customizing the labels on both axes
labs(x = "\nWeekdays", y = "Stream Counts per Weekday\n(Median)") +

# Label numbers with SI prefixes
scale_y_continuous(labels = label_number_si(accuracy = NULL)) +

# Customizing the legend
guides(color = guide_legend(override.aes = list(shape = 15, size = 4))) +

# Layout
theme_bw(base_size = 14) +
theme(axis.title.y = element_text(size = 14),
      axis.title.x = element_text(size = 14),
      axis.text.x = element_text(vjust = 0.5),
      legend.position = "top",
      legend.key = element_rect(color = "white"),
      legend.background = element_rect(fill = "white",
```

```
                                        linetype = "solid",
                                        color = "#bdbdbd"),
      plot.margin = unit(c(.66,.33,.66,.33), "cm"))

# Saving the output (wide format)
ggsave("weeks_scatter.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
        width = 21.33, height = 9, dpi = 320)
```

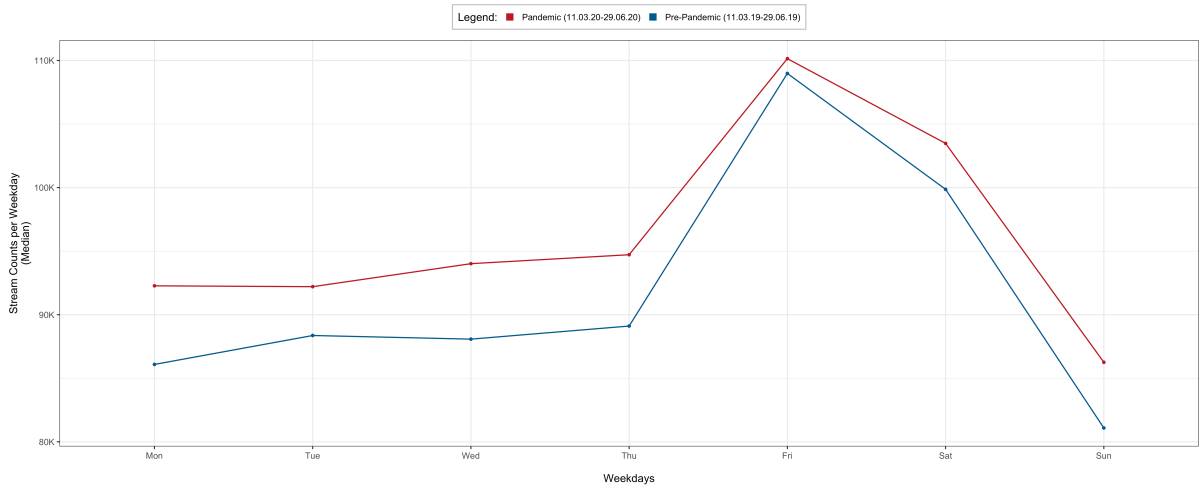

Figure 2: Connected Scatter plot: Stream counts (Median) of Germany's daily Top 200 Spotfy-charts within a Week Interval before and during the Pandemic.

That is, we have here no reason to accept and, respectively, retain the null hypothesis of the t-test, whose assumption purports that *no* significant differences in the proportions of the central tendencies of the compared sets are detectable at all (i.e., an assumption that the difference between the compared data points of both sets are symmetrically and, respectively, identically scattered around the respective means).

Next, we will approach the core of this study by preparing the entire dataset for a prediction task. By doing so, we finally scrutinize the second research hypothesis H_2.

3 Towards a Prediction Model

As we have detected a significant difference between those two periods after performing a t-Test and estimated a strong effect size after Cohen (1992) of this particular difference, the question now is this:

To what extent can we estimate and, respectively, predict the daily median stream counts of the pandemic period based on the whole pre-pandemic daily median stream counts?

To answer this question and to finally examine the research hypothesis H_2, we should create a new object that only consists of the dates and the median stream counts of each day from the entire dataset. This can be done with these code-lines:

```
spotifyRRR <- spotifyRR %>%
              group_by(date) %>%
              summarize(MdnStreams = median(stream_count))
```

Now, to see how these median stream counts are scattered within the whole period between 01.01.2019 and 29.06.2020, we should create once again a connected scatter plot as we have already done it for the stream counts data within a week interval, but now for the entire period.

```
spotifyRRR %>%

ggplot(aes(date, MdnStreams))+
```

```
# Connected Scatter plot
geom_point(size = .5)+
geom_line()+

# Annotating
geom_vline(xintercept = as.numeric(as.Date("2020-03-11")), color = "#252A52",
           size = 0.2, lty ="dashed")+

geom_label(aes(x = as.Date("2019-07-01"), y = 175e3, label = "Pre-Pandemic"),
           color = "#045a8d", size = 3)+

geom_label(aes(x = as.Date("2020-05-02"), y = 175e3,
               label = "Pandemic\ncontact restrictions"), color= "#bd0026",
           size = 3)+

# Detour: Extracting the date of the max stream count in December
#
# which.max(spotifyRRR$MdnStreams)
# [1] 358
#
# spotifyRRR$date[358]
# [1] "2019-12-24"
#
# spotifyRRR$MdnStreams[358]
# [1] 263718.5
#
# Row number 358 (i.e., Christmas) consists the maximal stream counts per day
# with 263718.5 streaming counts (median).

annotate(geom = "point", x = as.Date("2019-12-24"), y = 263718.5 , size = 5,
         shape = c(22,23), fill = "darkgoldenrod2", alpha = .4)+

geom_label(aes(x = as.Date("2019-11-19"), y = 262e3,
               label = "More than 263K\nStreams (Median)\non Christmas"),
               color = "darkgoldenrod2", size = 3)+
# Annotating
annotate("rect", xmin = as.Date("2019-01-01"), xmax = as.Date("2020-03-11"),
          ymin = -Inf, ymax = Inf, alpha = .08, fill = "#045a8d")+
annotate("rect", xmin = as.Date("2020-03-11"), xmax = as.Date("2020-06-29"),
         ymin = -Inf, ymax = Inf, alpha = .08, fill = "#bd0026")+

# Setting breaks on the x-axis
scale_x_date(date_breaks = "1 month", date_labels = "%b %y")+

# Label numbers with SI prefixes
scale_y_continuous(labels = label_number_si(accuracy = NULL))+

# Customizing the labels on both axes
labs(x = "\nStream Date", y = "Stream Counts per Day\n(Median)\n")+

# Layout
theme_bw(base_size = 14)+
theme(axis.title.y = element_text(size = 14),
      axis.title.x = element_text(size = 14),
      plot.margin = unit(c(.66, .33, .33, .33), "cm"))

# Saving the output (wide format)
ggsave("p2.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
```

```
       width = 21.33, height = 9, dpi = 320)
```

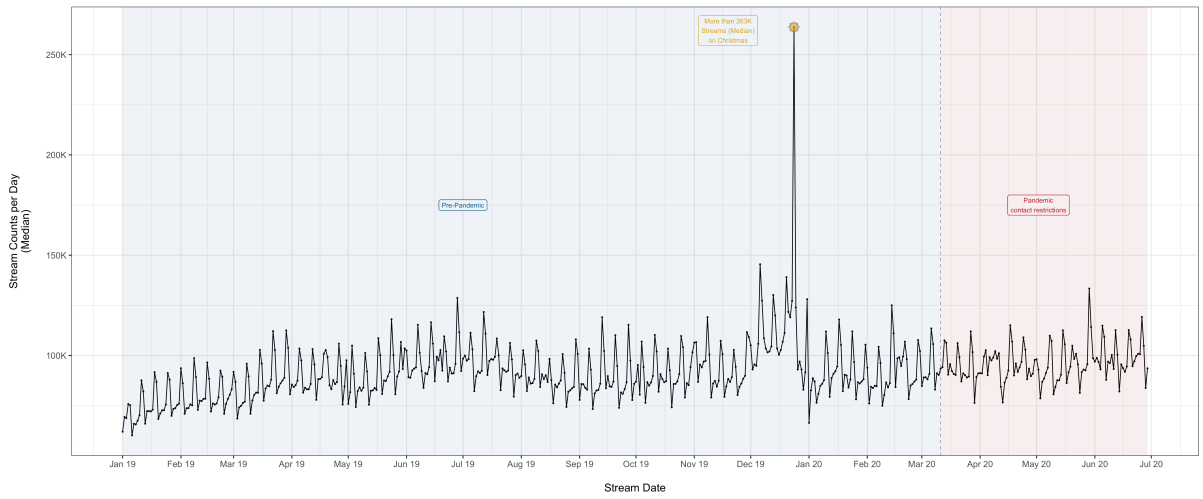

Figure 3: Connetected Scatter plot: Daily Stream Counts of Germany's Top 200 Spotify-charts (01.01.19—29.06.20).

As we see, there is a steep increase of stream counts in December with high peaks on Christmas and on New Year's Eve, which is not that surprising since people tend to use music as a tool for arousal and activation modulation or, respectively, to manage their moods (see also e.g., Juslin & Sloboda, 2011, for a profound overview of how people's arousal and activation levels in everyday life are influenced by the music they are listening to). During the rest of the year, the trend of the stream counts remains relatively stationary–with some minor seasonal dynamics. This issue matters now for answering our question. For if the data are not stationary, the time-series systematically influences the stream counts (i.e., the quantities would be in this case time-dependent such as it is the case in seasonal and/or trend periods, see (Hyndman & Athanasopoulos, 2018). That is, we need data whose properties are (almost) equal to white noise (i.e., a time series without serial correlations/autocorrelations). To test this assumption, we should use once again statistical tests. So for approaching this, we have to prepare the dataset a bit by converting it into time-series format because of a syntactic reason (R cannot handle the general `data.frame`-format in this regard):

```
# 1. From a data.frame into a tibble
spotifyR_tib <- tibble(Date = as.Date(spotifyRRR$date),
                       Actual_Stream_Counts = spotifyRRR$MdnStreams)

# 2. From a tibble into a tsibble
spotifyR_tsib <- as_tsibble(spotifyR_tib)

# 3. From a tsibble into a daily time-series, starting on the January 1st, 2019
spotifyR_TS <- as.ts(spotifyR_tsib, frequency = 365, start = c(2019,1))
```

Now we finally have a time-series dataset in the right format to continue with our analyses. Since we aim to estimate and, respectively, predict the daily median stream counts of the pandemic period based on the stream counts during the pre-pandemic period, we only evaluate the pre-pandemic period (time window) regarding its potential stationarity. This can be done as follows:

```
# The end-argument in the window-function indicates the 10. March (day 70 of the
# year 2020)

spotifyR_TS_orig <- window(spotifyR_TS, end = c(2020, 70))
```

As we assigned now the pre-pandemic period to an object, we will now have to determine the time period that we want to predict (i.e., the pandemic).

```
# We estimate the median stream counts of 111 days (that is the analyzed
# period or horizon of the COVID-19-pandemic of interest); start and end days
# are included.

horizon <- 111
```

Since we get predictions of the actual median stream counts with our trained models later on, we should compare these estimates with a benchmark model to be sure that our models and the invested workload are rewarding. For doing so, we now run a seasonal naïve model (see Hyndman & Athanasopoulos, 2018) that assumes that all values of the horizon period (i.e., 11.03.2020—29.06.2020) are equal to the values of the same period of the last observed season (i.e., in our case 11.03.2019—29.06.2019).

```
# Seasonal-naïve predicitons
benchmark <- snaive(spotifyR_TS_orig, h = horizon)

# Bringing the original data and the estimates together
spotifyR_tib_naïve <- spotifyR_tib %>%
mutate(SN_Estimates = c(rep(NA, length(spotifyR_TS_orig)), benchmark$mean)) %>%
tail(., 111)

# Calculating the benchmark accuracy
accuracy(spotifyR_tib_naïve$SN_Estimates,
         spotifyR_tib_naïve$Actual_Stream_Counts)
```

So with this, we have a benchmark model that should be outperformed with our models at stake. In particular, we should achieve with our models better error-metrics than an $RMSE = 13197.51$ and a lesser mean absolute percentage error of $MAPE = 11.19$. But before building the more advanced models, we should take a look at the estimates of this benchmark model:

```
ggplot(data = spotifyR_tib_naïve) +

# Connected scatter plot
geom_point(aes(Date, Actual_Stream_Counts),color = "black", size = .75) +
geom_line(aes(Date, Actual_Stream_Counts),color = "black", lty = "dashed") +

geom_point(aes(Date, SN_Estimates),color = "deeppink", size = .75) +
geom_line(aes(Date, SN_Estimates),color = "deeppink", lty = "solid") +

# Layout/Background
annotate("rect", xmin = as.Date("2020-03-11"), xmax = as.Date("2020-06-29"),
         ymin = -Inf, ymax = Inf, alpha = 0.08, fill = "#ef8a62") +

# Limiting the range of the x-axis to the pandemic period
scale_x_date(limits = c(as.Date("2020-03-11"), as.Date("2020-06-29")),
             date_breaks = "7 day", date_labels = "%d. %b %y") +

# Label numbers with SI prefixes
scale_y_continuous(labels = label_number_si(accuracy = NULL)) +

# Customizing the labels on both axes
labs(x = "\nStream Date", y = "Stream Counts per Day\n(Median)\n") +

# Layout
theme_bw(base_size = 14) +
theme(axis.title.y = element_text(size = 14),
      axis.title.x = element_text(size = 14),
      legend.position = "top",
      legend.key = element_rect(color = "white"),
      legend.background = element_rect(fill = "white",
```

```
                                        linetype = "solid",
                                        color = "#bdbdbd"),
      plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Saving the output (wide format)
ggsave("p3.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
       width = 21.33, height = 9, dpi = 320)
```

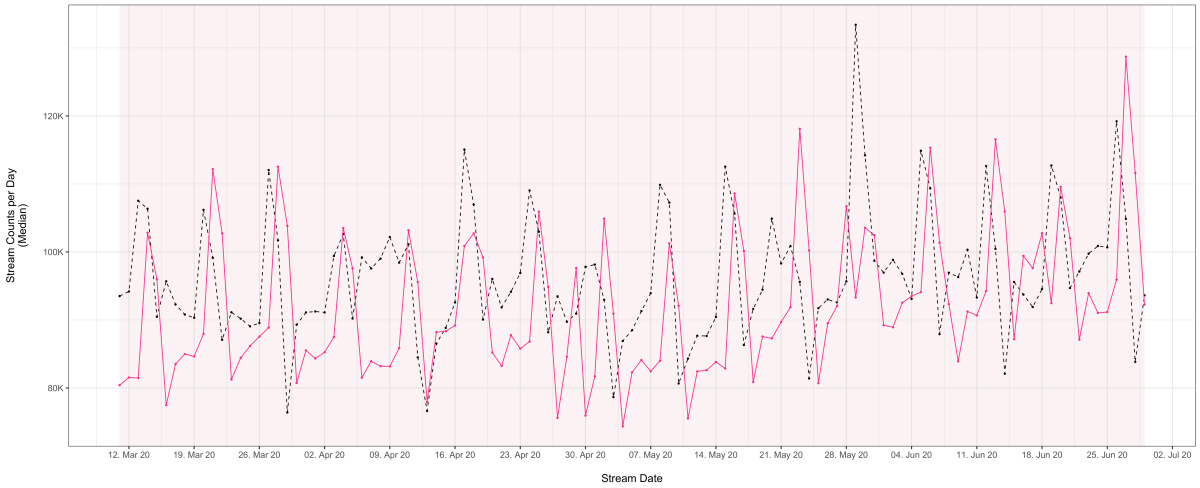

Figure 4: Connected Scatter plot: Comparison of Actual Stream Counts and Seasonal-Naïve Fits.

Now we should examine whether the original data follows a trend and whether the data consists of any seasonal pattern. Based on the plots from above, we can assume that the original data is not yet stationary since weekly pattern are clearly observable. If so, we should difference the data so the predictions base on the differences in values between the time steps and not on the value itself—with this, we can finally overcome possible autocorrelations (see for a more detailed introduction to this topic Hyndman & Athanasopoulos, 2018).

```
# Is the data stationary? If not: How many times does the data need to be
# differenced?

ndif <- ndiffs(spotifyR_TS_orig)
print(ndif)

# Apparently, we have to difference our data only once. This appears reasonable
# based on the inspection of the connected scatter plot from above.

# For doing so, we log-transform the data, then we difference the data
# (i.e., stabilizing the variance and the means; also known as de-trending and
# de-seasonalizing)

spotifyR_TS_trans <- diff(log(spotifyR_TS_orig), differences = ndif)

# How many lags (i.e., repeating frequency pattern) does the period consist?
#
lag_selection <- vars::VARselect(spotifyR_TS_trans)

print(lag_selection$selection)

# Lag of 7 seems reasonable since we can see in the connected scatter plot a
# weekly pattern: 4-5 spikes per month (= 4-5 pattern of 7 days)
```

```
lags <- 7
```

Next, we will test whether the data follows a trend (time-dependency) by using the so-called Augmented Dickey Fuller-Test whose null hypothesis states that the data has a unit-root (see for a brief introduction Holmes, Scheuerell, Ward, 2020).

```
# ADF: Null hypo: Data has a unit-root. That is, we want reject the null
# (i.e., p < .05)

spotifyR_TS_trans %>%
# we choose "none" since we already have transformed the data
ur.df(lags = lags, type = "none") %>%
summary()
```

Based on this test result, we can reasonably reject the null hypothesis. That is, our data does not have unit-root according to the result of the augmented Dickey-Fuller-Test ($ADF_{Test-statistic} = -9.046$, $\tau_1 = -1.95$, $p < .001$, $CL = 95\%$). Hence, we can reasonably conclude that the data seems to be stationary.

```
spotifyR_TS_trans %>%

# Here we can use the autoplot-function since it provides everything we need.
# Otherwise, we had to convert (and manipulate) the time series-object to a
# data.frame-format. This would be, however, unnecessary additional work.

autoplot() +
geom_point(size = .5) +

# Annotating
geom_label(aes(x = decimal_date(as.Date(c("2019-07-01"))), y = 0.5,
               label = "Pre-Pandemic"), color = "#045a8d",
               size = 3) +

# Layout/Background
annotate("rect", xmin = 2019, xmax = decimal_date(as.Date(c("2020-03-10"))),
          ymin = -Inf, ymax = Inf, alpha = .08, fill = "#045a8d") +

# Customizing the labels on both axes
labs(x = "\nStream Period",
     y = "Stream Counts per Day\n(log-transformed and differenced)\n") +

# Layout
theme_bw(base_size = 14) +
theme(axis.title.y = element_text(size = 14),
      axis.title.x = element_text(size = 14),
      plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Saving the plot (wide format)
ggsave("p4.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
       width = 21.33, height = 9, dpi = 320)
```

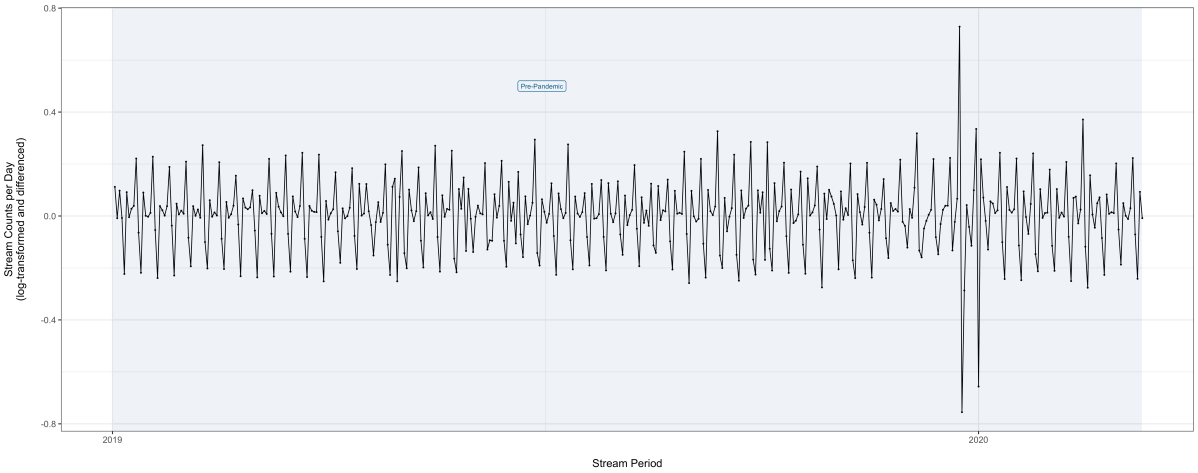

Figure 5: Stationarized Pre-Pandemic Period (7 Lags).

Now, since we aim to know to what extent the daily median stream counts during the first wave of the COVID-19 pandemic in Germany are predictable based on the stream counts of the pre-pandemic period, we can interpret this prediction-problem within the framework of supervised machine learning-techniques.

To predict now the daily median stream counts during the first wave of the COVID-19 pandemic in Germany, we will use two well-known algorithms. The first of which is the Random Forest algorithm, the second of which is the k-Nearest Neighbors (kNN) algorithm (see for a general introduction to these algorithms James et al., 2015).

The reasons for choosing these algorithms are mainly of pragmatic nature: Since the Random Forest provides usually not only less errors in prediction and classification tasks but is also robust and relatively easy to train and computationally not that expensive, we will use this approach. To compare the performance of this algorithm with another advanced approach than the season-naïve estimates, we will use the kNN algorithm as a so-called "lazy learner" (Rhys, 2020, p. 284) in contrast to the Random Forest algorithm.

3.1 Time Delay Embedding

At this point, we approach one of the most crucial parts of our analyses since our prediction will rely on the following considerations. We embed our transformed time series into a low-dimensional Euclidean space (this is essentially a two-dimensional coordinate system) to benefit from a two-dimensional data structure (i.e., matrix). That is, we create a matrix in which the suggested dimensions of lags (according to AIC; in our case 7, see for this Information Criterion e.g., James et al., 2015, p. 212) are considered as predictors for the remaining column of original index-data points. Thus, we get with this a matrix with eight columns where the first column consists the original data points. The other seven columns consist of those streaming count values that are being proportionally shifted or delayed predictors for the first column. For example, if the first column consists five rows, and if we have three predictors and add one dimension to them, then we will get a matrix with two rows and four columns because every row of the original column is now shifted by one predictor dimension (see e.g. this paper von Oertzen & Boker (2010)).

```
# An example:

x <- seq(1:5)
x %<>% as.matrix()

# A usual matrix:

>print(x)
      [,1]
```

```
 [1,]    1
 [2,]    2
 [3,]    3
 [4,]    4
 [5,]    5
>

# A matrix with embedded dimensions: 3 Lags + 1

>embed(x, 3 + 1)

     [,1] [,2] [,3] [,4]
[1,]    4    3    2    1
[2,]    5    4    3    2
>
```

We apply now this function to our time series data to get the mentioned matrix:

```
spotifyR_TS_trans_embed <- embed(spotifyR_TS_trans,lags + 1)
```

3.2 Splitting the Dataset

Before we can train our models, we have to create training and test sets. Since we want to predict or, respectively, estimate the stream counts of the pandemic period based on the stream counts of the pre-pandemic period, we cannot use the pandemic period for training purposes. Thus, the pandemic period represents our already given test set, whereas the pre-pandemic period poses the training set.

```
# Regressand (aka: target, label, dependent variable)

y_train <-  spotifyR_TS_trans_embed[, 1]

# Regressor (aka: input, independent variable)

X_train <- spotifyR_TS_trans_embed[, -1]

# The actual test set with those suggested lags

X_valid <- spotifyR_TS_trans_embed[nrow(spotifyR_TS_trans_embed), c(1:lags)]

# Now we assign the actual final dataset that we want to estimate with our
# models. In our case: The time series between March 11th, 2020 (day 71 of the year
# 2020) and June 29th, 2020 (day 181 of the year 2020); this is the examined period
# of the COVID-19 pandemic.

y_test <- window(spotifyR_TS, start = c(2020, 71), end = c(2020, 181))
```

4 Training the Models

We are close, however, we should, optimize certain parameters (so-called hyperparameters) of the algorithms to make sure that we train models with, so to speak, good pre-conditions. Particularly, we are going to run 10-fold cross-validations to identify the optimal hyperparameters of the respective algorithms. This can be done with the following approach:

```
# 10-fold cross-validation
rf_train_control <- trainControl(method = "repeatedcv", number = 10,
                                 repeats = 5)

# We define the number of predictors in the mtry-object. This is the number
# of randomly chosen splits at each tree. According to the suggestion of
```

```r
# Breiman and Cutler (2018) we should divide the predictors by 3 for
# regression-approaches.

rf_tune_grid <- expand.grid(mtry = col(X_train/3),
                            # using extra trees as random split points instead
                            # of bootstrapped samples. Each tree uses the whole
                            # training data.
                            splitrule = "extratrees",
                            # tree complexity: default 5 for regressions
                            min.node.size = 5)

set.seed(61, sample.kind = "Rounding")

rf_kfold_cv <- train(data.frame(X_train), y_train,
                     # faster Random Forest implementation
                     method = "ranger",
                     trControl = rf_train_control, tuneGrid = rf_tune_grid)

print(rf_kfold_cv$bestTune[,1])
################################################################################

# 10-fold cross-validation
knn_train_control <- trainControl(method = "repeatedcv", number = 10,
                                  repeats = 5)

# Determining the optimal number of neighbors of data points with which the next
# data point is estimated based on the resulting outcome values of the
# nearest neighbors on average. Note, that the "knn" method of the caret-package
# provides here "regression" error metrics. The "best" k value provides here the
# lowest RMSE value.

knn_tune_grid <- expand.grid(k = c(1:25))

set.seed(61, sample.kind = "Rounding")

knn_kfold_cv <- train(data.frame(X_train), y_train, method = "knn",
                      trControl = knn_train_control, tuneGrid = knn_tune_grid)
print(knn_kfold_cv$bestTune[,1])
```

4.1 Training-Loops

Finally, we can now train our models with optimized hyperparameters by looping the models through every horizon point (i.e., 111 days). That is, we essentially compute 111 tiny sub-models. One model for each data point of the horizon.

```r
# Now, we save each estimate in a blank object (like a container that gets
# filled)

rf_estimates <- numeric(horizon)

set.seed(1, sample.kind = "Rounding")

# For-loop: Random Forest Training

for (i in 1:horizon) {

set.seed(1857, sample.kind = "Rounding")
```

```
rf_mod <- randomForest(X_train, y_train, mtry = rf_kfold_cv$bestTune[,1],
                       nodesize = 5, replace = T, splitrule = "extratrees")

# Here we fill the mentioned container with the predicted values

rf_estimates[i] <- predict(rf_mod, X_valid)

# The training data is here constantly updated to account for the time distance
# corresponding to the respective data point of the horizon.

y_train <- y_train[-1]

X_train <- X_train[-nrow(X_train),] }

#################################################################################

# Setting back the training objects. Ensuring a fresh start.

y_train <- spotifyR_TS_trans_embed[, 1]
X_train <- spotifyR_TS_trans_embed[, -1]

# Same procedure as above, however, adjusted to the kNN-algorithm

knn_estimates <- numeric(horizon)

# For-loop: kNN Training

set.seed(1, sample.kind = "Rounding")

for (j in 1:horizon) {

set.seed(1857, sample.kind = "Rounding")

knn_estimates[j] <- knn.reg(train = data.frame(X_train), test = X_valid,
                            y = y_train, k = knn_kfold_cv$bestTune[,1])$pred

y_train <- y_train[-1]

X_train <- X_train[-nrow(X_train),] }
```

Having now trained our models, we need to retransform the predicted values back to their original scale.

4.2 Retransforming Estimates

Since we transformed the original data, we now need to retransform it to have meaningful values. For doing so, we take at first the cumulative sum of our transformed estimates. Then we take of these cumulative estimates the exponents (i.e., the opposite of the previous logarithms). To finally get the values in their original format, we simply multiply this result with the last observation of the original data. That is, we calculate this

$$y_t \exp(\sum_{i=1}^{h} \tilde{y}_t + i)$$

where y_t is the original data and h indicates the horizon and i its index, while $\tilde{y}_t + i$ stands for the estimates.

```
# Retransforming the estimates by taking the "anti-logs" (i.e., computing
# exponents for every estimate).
```

```
rf_exponents_of_estimates <- exp(cumsum(rf_estimates))
knn_exponents_of_estimates <- exp(cumsum(knn_estimates))

# Extracting the last observation from the time-series.

last_observation <- as.vector(tail(spotifyR_TS_orig, 1))

# Getting the final values by retransforming them.

rf_retransformed_estimates <- last_observation * rf_exponents_of_estimates
knn_retransformed_estimates <- last_observation * knn_exponents_of_estimates

# Converting them into time series-format

rf_y_pred <- ts(rf_retransformed_estimates, start = c(2020, 71),
                frequency = 365)
knn_y_pred <- ts(knn_retransformed_estimates, start = c(2020, 71),
                 frequency = 365)
```

Once we have retransformed our data, we are now ready to evaluate the results of our predictions.

5 Test Results

Now, we have approached the most thrilling part of our endeavor. We will now see the error metrics. This is a very crucial part with regard to our hypothesis. For if the error values are bad (high) for our models, we have failed to predict the stream counts appropriately.

```
# Bringing the original data and the estimates together

spotifyR_tib_star <- spotifyR_tib %>%
mutate(RF_Estimates = c(rep(NA, length(spotifyR_TS_orig)), rf_y_pred),
       kNN_Estimates = c(rep(NA, length(spotifyR_TS_orig)), knn_y_pred),
       SN_Estimates = c(rep(NA, length(spotifyR_TS_orig)), benchmark$mean))

# Extracting only the pandemic-period

test <-  tail(spotifyR_tib_star, horizon)

# Calculating the error metrics

fuzzyr.accuracy(test$RF_Estimates, test$Actual_Stream_Counts, test$SN_Estimates)

fuzzyr.accuracy(test$kNN_Estimates, test$Actual_Stream_Counts,test$SN_Estimates)
```

Now, as the tension was great, we can now calm down since we have got some great results ($RMSE_{RF} = 6633.31$, $MAPE_{RF} = 5.29$; $RMSE_{kNN} = 6926.32$, $MAPE_{kNN} = 5.60$) and, besides that: Our model performs a way better (49% more accuracy) than the seasonal-naïve method ($UMBRAE_{RF:SN} = 0.51$; see for the UMBRAE accuracy metric: Chen, Twycross, Garibald, 2017). This is pretty satisfying. But as usual, we should really plot these results to get a concrete impression of how our predicted stream count values fit to the actual stream counts. In this vein, it is indeed helpful to compare the results with each other. However, to see to what extent the predictions of the best model (i.e., the Random Forest model) are lying within the prediction interval, we should first calculate these prediction intervals (see Hyndman & Athanasopoulos, 2018).

```
resids <- test$Actual_Stream_Counts - test$RF_Estimates
shapiro.test(resids)

# Since the residuals are normally distributed according to the Shapiro-Wilk-
```

```
# Test (W = 0.98609, p = 0.307), we can calculate the prediction intervals for
# each point by multiplying the standard deviation of the residuals with the
# z-score of the 0.025th and, respectively, the 0.975th percentile point.

SD_star <- sd(resids)

spotifyR_tib_star %<>%
mutate(RF_E_lo.95 = RF_Estimates  - (1.96 * SD_star),
       RF_E_hi.95 = RF_Estimates  + (1.96 * SD_star))
```

Now it is time to plot the final outcome of our endeavor in a connected scatter plot with the computed range of prediction intervals for each estimate of the Random Forest model.

```
# Creating a color-object

cols <- c("Actual Stream Counts (Median)" = "black",
          # Colorblind Green
          "Random Forest Predictions\nwithin 95% PI" = "#009E73",
          # Colorblind Orange
          "kNN Predictions" = "#E69F00",
          # Colorblind Pink
          "Seasonal-naïve Predictions" = "#CC79A7")

spotifyR_tib_star %>%
filter(Date >= as.Date("2020-03-11")) %>%
ggplot(aes(x = Date)) +

# Connected scatter plot
geom_point(aes(y = Actual_Stream_Counts,
               color = "Actual Stream Counts (Median)"), size = .75) +
geom_line(aes(y = Actual_Stream_Counts,
              color = "Actual Stream Counts (Median)"), lty = "dashed") +

geom_point(aes(y = RF_Estimates,
               color = "Random Forest Predictions\nwithin 95% PI"),
              size = .75) +
geom_line(aes(y = RF_Estimates,
               color = "Random Forest Predictions\nwithin 95% PI" ),
               lty = "solid") +

geom_point(aes(y = kNN_Estimates, color = "kNN Predictions"), size = .75) +
geom_line(aes(y = kNN_Estimates, color = "kNN Predictions"),lty = "solid") +

geom_point(aes(y = SN_Estimates, color = "Seasonal-naïve Predictions"),
              size = .75) +
geom_line(aes(y = SN_Estimates,color = "Seasonal-naïve Predictions"),
              lty = "solid") +

# Prediction Interval
geom_ribbon(aes(y = RF_Estimates, ymin = RF_E_lo.95, ymax = RF_E_hi.95),
  fill = "#009E73", alpha = 0.2) +

geom_vline(xintercept = as.numeric(as.Date("2020-03-11")),
  color = "#0072B2", size = 0.2, lty = "dashed") +
geom_label(aes(x = as.Date("2020-05-03"), y = 13e4,
  label = "Test Dataset\n(Pandemic Period)"  ),  color = "#ef8a62",
  size = 6) +
annotate("rect",  xmin = as.Date("2020-03-11"), xmax = as.Date("2020-06-29"),
```

```
        ymin = -Inf, ymax = Inf, alpha = 0.08, fill = "#ef8a62") +

# Setting breaks on the x-axis
scale_x_date(limits = c(as.Date("2020-03-11"), as.Date("2020-06-29")),
             date_breaks = "7 day", date_labels = "%d. %B %y") +

# Label numbers with SI prefixes
scale_y_continuous(labels = label_number_si(accuracy = NULL)) +

# Legend colors
scale_colour_manual(name = "Legend:", values = cols) +
scale_fill_manual(name = "Legend:", values = cols) +

# Customizing the labels on both axes
labs(x = "\nDate of Streams", y = "Stream Counts per Day\n(Median)\n") +

# Layout
theme_bw(base_size = 14) +
theme(axis.title.y = element_text(size = 14),
      axis.title.x = element_text(size = 14, vjust = -1),
      legend.position = "top",
      legend.key = element_rect(color = "white"),
      legend.background = element_rect(fill = "white",
                                       linetype = "solid",
                                       color = "#bdbdbd"),
      plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Saving the output (wide format)
ggsave("p6.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
       width = 21.33, height = 9, dpi = 320)
```

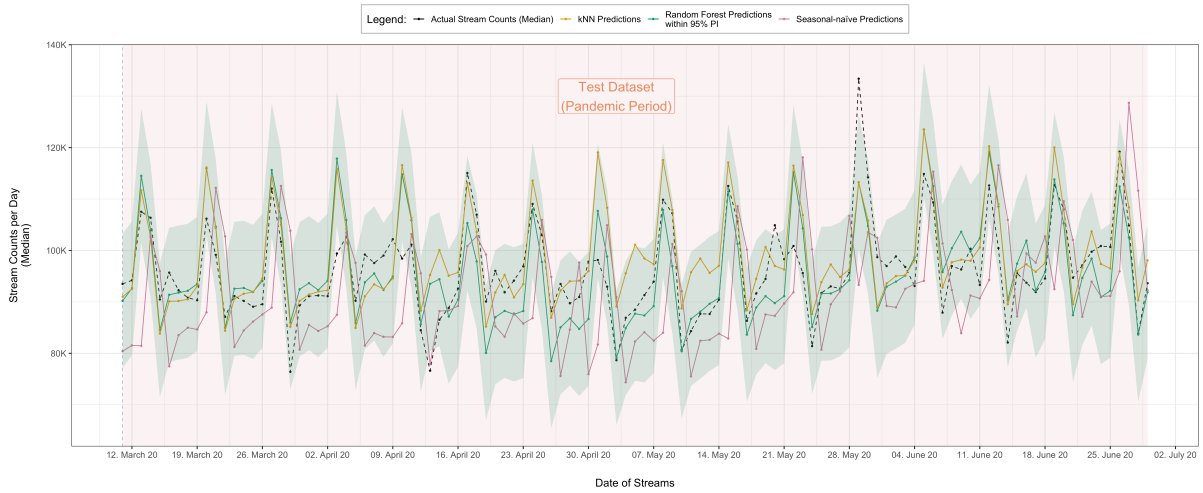

Figure 6: Connected Scatter plot: Comparison of actual streaming counts, Random Forest-estimates, and seasonal-naïve-fits. The green ribbon represents approximated SD based on the residuals of the Random Forest-estimates of the actual stream counts

Conclusion

Finally, we see the results of all three prediction-approaches of the daily median stream counts during the first wave of the COVID-19 pandemic in Germany together. We can state that all in all the actual stream counts and our Random Forest-estimates go along—with some deviations (see for example, April 13th).

At the end of our endeavor we can answer our initial question with the tested hypothesis now as follows: People tend to listen to more music during the COVID-19 pandemic. At least, there are significant and strong differences within a week interval detectable. Hence, we can accept our first hypothesis. On the other hand, we could also predict the median stream counts as based on dates with Random Forest and kNN-regressions with desirable error metrics. Possible next steps could be to run a sentiment analysis on the selected songs and tracks and/or to examine to what extent these sentiments are related with audio features of the respective tracks. With our analyses we could after all show that at least stream counts are predictable based on dates from a supervised machine learning perspective.

Appendix — Environment

Overview of the used statistical computing environment and the used operating system:

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 4.0.3 (2020-10-10)
##  os       macOS Big Sur 10.16
##  system   x86_64, darwin17.0
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       Europe/Berlin
##  date     2021-01-14
##
## - Packages -------------------------------------------------------------------
##  package      * version    date       lib source
##  anytime        0.3.9      2020-08-27 [2] CRAN (R 4.0.2)
##  assertthat     0.2.1      2019-03-21 [2] CRAN (R 4.0.0)
##  backports      1.2.0      2020-11-02 [2] CRAN (R 4.0.2)
##  broom          0.7.2      2020-10-20 [2] CRAN (R 4.0.2)
##  caret        * 6.0-86     2020-03-20 [2] CRAN (R 4.0.0)
##  cellranger     1.1.0      2016-07-27 [2] CRAN (R 4.0.0)
##  class          7.3-17     2020-04-26 [2] CRAN (R 4.0.3)
##  cli            2.2.0      2020-11-20 [2] CRAN (R 4.0.2)
##  codetools      0.2-16     2018-12-24 [2] CRAN (R 4.0.3)
##  colorspace     2.0-0      2020-11-11 [2] CRAN (R 4.0.2)
##  crayon         1.3.4      2017-09-16 [2] CRAN (R 4.0.0)
##  curl           4.3        2019-12-02 [2] CRAN (R 4.0.0)
##  data.table     1.13.2     2020-10-19 [2] CRAN (R 4.0.2)
##  DBI            1.1.0      2019-12-15 [2] CRAN (R 4.0.0)
##  dbplyr         2.0.0      2020-11-03 [2] CRAN (R 4.0.2)
##  digest         0.6.27     2020-10-24 [2] CRAN (R 4.0.2)
##  dplyr        * 1.0.2      2020-08-18 [2] CRAN (R 4.0.2)
##  e1071          1.7-4      2020-10-14 [2] CRAN (R 4.0.2)
##  ellipsis       0.3.1      2020-05-15 [2] CRAN (R 4.0.0)
##  evaluate       0.14       2019-05-28 [2] CRAN (R 4.0.0)
##  expsmooth    * 2.3        2015-04-09 [2] CRAN (R 4.0.0)
##  fansi          0.4.1      2020-01-08 [2] CRAN (R 4.0.0)
##  fastmap        1.0.1      2019-10-08 [2] CRAN (R 4.0.0)
##  fma          * 2.4        2020-01-14 [2] CRAN (R 4.0.0)
##  FNN          * 1.1.3      2019-02-15 [2] CRAN (R 4.0.2)
##  forcats      * 0.5.0      2020-03-01 [2] CRAN (R 4.0.0)
##  foreach        1.5.1      2020-10-15 [2] CRAN (R 4.0.2)
##  forecast     * 8.13       2020-09-12 [2] CRAN (R 4.0.2)
##  fpp2         * 2.4        2020-09-09 [2] CRAN (R 4.0.2)
##  fracdiff       1.5-1      2020-01-24 [2] CRAN (R 4.0.0)
##  fs             1.5.0      2020-07-31 [2] CRAN (R 4.0.2)
```

```
##  FuzzyR       * 2.3        2019-11-20 [2] CRAN (R 4.0.0)
##  generics       0.1.0      2020-10-31 [2] CRAN (R 4.0.2)
##  ggExtra      * 0.9        2019-08-27 [2] CRAN (R 4.0.2)
##  ggplot2      * 3.3.2      2020-06-19 [2] CRAN (R 4.0.0)
##  ggrepel      * 0.8.2      2020-03-08 [2] CRAN (R 4.0.0)
##  glue           1.4.2      2020-08-27 [2] CRAN (R 4.0.2)
##  gower          0.2.2      2020-06-23 [2] CRAN (R 4.0.2)
##  gtable         0.3.0      2019-03-25 [2] CRAN (R 4.0.0)
##  haven          2.3.1      2020-06-01 [2] CRAN (R 4.0.2)
##  hms            0.5.3      2020-01-08 [2] CRAN (R 4.0.0)
##  htmltools      0.5.0      2020-06-16 [2] CRAN (R 4.0.2)
##  httpuv         1.5.4      2020-06-06 [2] CRAN (R 4.0.0)
##  httr           1.4.2      2020-07-20 [2] CRAN (R 4.0.2)
##  ipred          0.9-9      2019-04-28 [2] CRAN (R 4.0.0)
##  iterators      1.0.13     2020-10-15 [2] CRAN (R 4.0.2)
##  jsonlite       1.7.1      2020-09-07 [2] CRAN (R 4.0.2)
##  kableExtra   * 1.3.1      2020-10-22 [2] CRAN (R 4.0.2)
##  knitr        * 1.30       2020-09-22 [2] CRAN (R 4.0.2)
##  later          1.1.0.1    2020-06-05 [2] CRAN (R 4.0.0)
##  lattice      * 0.20-41    2020-04-02 [2] CRAN (R 4.0.3)
##  lava           1.6.8.1    2020-11-04 [2] CRAN (R 4.0.2)
##  lifecycle      0.2.0      2020-03-06 [2] CRAN (R 4.0.0)
##  lmtest         0.9-38     2020-09-09 [2] CRAN (R 4.0.2)
##  lubridate    * 1.7.9.2    2020-11-13 [2] CRAN (R 4.0.2)
##  magrittr     * 2.0.1      2020-11-17 [2] CRAN (R 4.0.2)
##  MASS           7.3-53     2020-09-09 [2] CRAN (R 4.0.3)
##  Matrix         1.2-18     2019-11-27 [2] CRAN (R 4.0.3)
##  mime           0.9        2020-02-04 [2] CRAN (R 4.0.0)
##  miniUI         0.1.1.1    2018-05-18 [2] CRAN (R 4.0.0)
##  ModelMetrics   1.2.2.2    2020-03-17 [2] CRAN (R 4.0.0)
##  modelr         0.1.8      2020-05-19 [2] CRAN (R 4.0.2)
##  munsell        0.5.0      2018-06-12 [2] CRAN (R 4.0.0)
##  nlme           3.1-149    2020-08-23 [2] CRAN (R 4.0.3)
##  nnet           7.3-14     2020-04-26 [2] CRAN (R 4.0.3)
##  pillar         1.4.7      2020-11-20 [2] CRAN (R 4.0.2)
##  pkgconfig      2.0.3      2019-09-22 [2] CRAN (R 4.0.0)
##  plyr           1.8.6      2020-03-03 [2] CRAN (R 4.0.0)
##  pROC           1.16.2     2020-03-19 [2] CRAN (R 4.0.0)
##  prodlim        2019.11.13 2019-11-17 [2] CRAN (R 4.0.0)
##  promises       1.1.1      2020-06-09 [2] CRAN (R 4.0.2)
##  purrr        * 0.3.4      2020-04-17 [2] CRAN (R 4.0.0)
##  quadprog       1.5-8      2019-11-20 [2] CRAN (R 4.0.0)
##  quantmod       0.4.17     2020-03-31 [2] CRAN (R 4.0.2)
##  R6             2.5.0      2020-10-28 [2] CRAN (R 4.0.2)
##  randomForest * 4.6-14     2018-03-25 [2] CRAN (R 4.0.0)
##  ranger       * 0.12.1     2020-01-10 [2] CRAN (R 4.0.0)
##  Rcpp           1.0.5      2020-07-06 [2] CRAN (R 4.0.2)
##  readr        * 1.4.0      2020-10-05 [2] CRAN (R 4.0.2)
##  readxl         1.3.1      2019-03-13 [2] CRAN (R 4.0.0)
##  recipes        0.1.15     2020-11-11 [2] CRAN (R 4.0.2)
##  reprex         0.3.0      2019-05-16 [2] CRAN (R 4.0.0)
##  reshape2       1.4.4      2020-04-09 [2] CRAN (R 4.0.0)
##  rlang          0.4.9      2020-11-26 [1] CRAN (R 4.0.2)
##  rmarkdown      2.5        2020-10-21 [2] CRAN (R 4.0.2)
##  rpart          4.1-15     2019-04-12 [2] CRAN (R 4.0.3)
##  rstudioapi     0.13       2020-11-12 [2] CRAN (R 4.0.2)
##  rvest        * 0.3.6      2020-07-25 [2] CRAN (R 4.0.2)
##  sandwich       3.0-0      2020-10-02 [2] CRAN (R 4.0.2)
```

```
##  scales       * 1.1.1      2020-05-11 [2] CRAN (R 4.0.0)
##  sessioninfo    1.1.1      2018-11-05 [2] CRAN (R 4.0.0)
##  shiny          1.5.0      2020-06-23 [2] CRAN (R 4.0.2)
##  stringi        1.5.3      2020-09-09 [2] CRAN (R 4.0.2)
##  stringr      * 1.4.0      2019-02-10 [2] CRAN (R 4.0.0)
##  strucchange    1.5-2      2019-10-12 [2] CRAN (R 4.0.0)
##  survival       3.2-7      2020-09-28 [2] CRAN (R 4.0.3)
##  tibble       * 3.0.4      2020-10-12 [2] CRAN (R 4.0.2)
##  tidyr        * 1.1.2      2020-08-27 [2] CRAN (R 4.0.2)
##  tidyselect     1.1.0      2020-05-11 [2] CRAN (R 4.0.0)
##  tidyverse    * 1.3.0      2019-11-21 [2] CRAN (R 4.0.0)
##  timeDate       3043.102   2018-02-21 [2] CRAN (R 4.0.0)
##  tseries        0.10-47    2019-06-05 [2] CRAN (R 4.0.0)
##  tsibble      * 0.9.3      2020-10-06 [2] CRAN (R 4.0.2)
##  TTR            0.24.2     2020-09-01 [2] CRAN (R 4.0.2)
##  urca         * 1.3-0      2016-09-06 [2] CRAN (R 4.0.0)
##  vars           1.5-3      2018-08-06 [2] CRAN (R 4.0.0)
##  vctrs          0.3.5      2020-11-17 [2] CRAN (R 4.0.2)
##  viridisLite    0.3.0      2018-02-01 [2] CRAN (R 4.0.0)
##  webshot        0.5.2      2019-11-22 [2] CRAN (R 4.0.2)
##  withr          2.3.0      2020-09-22 [2] CRAN (R 4.0.2)
##  xfun           0.19       2020-10-30 [2] CRAN (R 4.0.2)
##  xml2         * 1.3.2      2020-04-23 [2] CRAN (R 4.0.0)
##  xtable         1.8-4      2019-04-21 [2] CRAN (R 4.0.0)
##  xts            0.12.1     2020-09-09 [2] CRAN (R 4.0.2)
##  yaml           2.2.1      2020-02-01 [2] CRAN (R 4.0.0)
##  zoo            1.8-8      2020-05-02 [2] CRAN (R 4.0.2)
##
## [1] /Users/Kework/Library/R/4.0/library
## [2] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```