Spotify Streaming Before and During the First Wave of the COVID-19 Pandemic in Germany:

Compared and Predicted

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 or explicit 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 pandemic-related restrictions and limitations.

Building on these quite general observations and assumptions, we can raise the following general question:

- *RQ*: To what extent listen people more music during the COVID-19 pandemic compared with the period before the pandemic and how are daily music stream counts during the COVID-19 pandemic predictable?

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 (2021) provides data of such stream counts so that they can be retrieved via 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 in Germany tend to stream during the COVID-19 pandemic considerably more of Germany's daily top 200 Spotify charts than before.
- H_2: The stream counts of Germany's daily top 200 Spotify charts during the COVID-19 pandemic are predictable based on daily stream counts of the pre-pandemic period.

Since we now have formulated a general question that we aim to answer by testing those introduced hypotheses, and as we have narrowed down our interest as well, we can begin to explore the stream 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 install some packages, which consist of the required functions for the analyses in question. 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",
                      # Date wrangling
                      "lubridate",
```

```
                        # Graphics
                        "ggExtra", "ggrepel", "scales",
                        # Tidy Statistics
                        "rstatix",
                        # Machine Learning + additional kNN and Random Forest
                        # algorithms
                        "caret", "randomForest", "ranger", "FNN",
                        # Forecasting
                        "forecast", "fpp2",
                        # Time series related tests / tools / accuracy-metrics
                        "urca", "FuzzyR", "nonlinearTseries",
                        # Variance Analysis
                        "vars",
                        # 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.

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

After loading and installing the desired packages, we have to web-scrape data from the streaming service provider Spotify (2021) 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 546 days, and since each day consists of 200 chart positions, we will aggregate a final dataset with 109,200 rows (546 * 200) with the following lines (see Argueta, 2018, for the original proposal for the web-scraping procedure of Spotify charts):

```
# 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) {
```

```r
# Retrieving the 200 chart positions of each day.

chart_position <- read_html(x) %>%
                  html_elements(".chart-table-position") %>%
                  html_text()

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

title <- read_html(x) %>%
         html_elements("strong")%>%
         html_text()

# Retrieving the 200 artist names of each day.

artist <-read_html(x) %>%
          html_elements(".chart-table-track span")%>%
          html_text()

# Retrieving the 200 stream counts of each day.

stream_count <- read_html(x) %>%
                html_elements("td.chart-table-streams") %>%
                html_text()

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

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

# Putting these chunks together in a dataframe.

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(all_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 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 from my GitHub repo

spotifyR <- read_csv("https://git.io/JBCbd")
```

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 Dataset

chart_position	title	artist	stream_count	date
1	Benzema	by Capital Bra	439742	01/01/2019
2	HaifischNikez Allstars	by LX, Maxwell, Bonez MC, Gzuz, Sa4	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 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.

1.2 Data Cleaning

To clean 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 (cf., Footnote 1):

```
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(gsub(",","", stream_count)),

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

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 Dataset

chart_position	title	artist	stream_count	date	n	songID
1	Benzema	Capital Bra	439742	2019-01-01	340	316
2	HaifischNikez Allstars	LX, Maxwell, Bonez MC, Gzuz, Sa4	373845	2019-01-01	143	943
3	Sweet but Psycho	Ava Max	353209	2019-01-01	502	2302

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.

```
# Pre-Pandemic Period
Streams <- spotifyRR %>%
           filter(date <= as.Date("2020-03-10"))
# Pandemic Period
Streams_Pan <- spotifyRR %>%
               filter(date >= as.Date("2020-03-11"))

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.

png("dist_hist.png",  width = 12.8, height = 7.2,units ="in", res = 300 )
ggplot() +

# Frequency Histogram
geom_histogram(data = Streams, aes(stream_count,
                                   fill = "Pre-Pandemic (01.01.19-10.03.20)"),
               binwidth  = binw, alpha = 0.7, color ="darkgrey", size=0.005) +
```

```
geom_histogram(data = Streams_Pan, aes(stream_count,
                                       fill= "Pandemic (11.03.20-29.06.20)"),
               binwidth  = binw_Pan, alpha = 0.7, color="darkgrey", size=0.005) +

# 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 important 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 = 12.8, height = 7.2, dpi = 320)
```

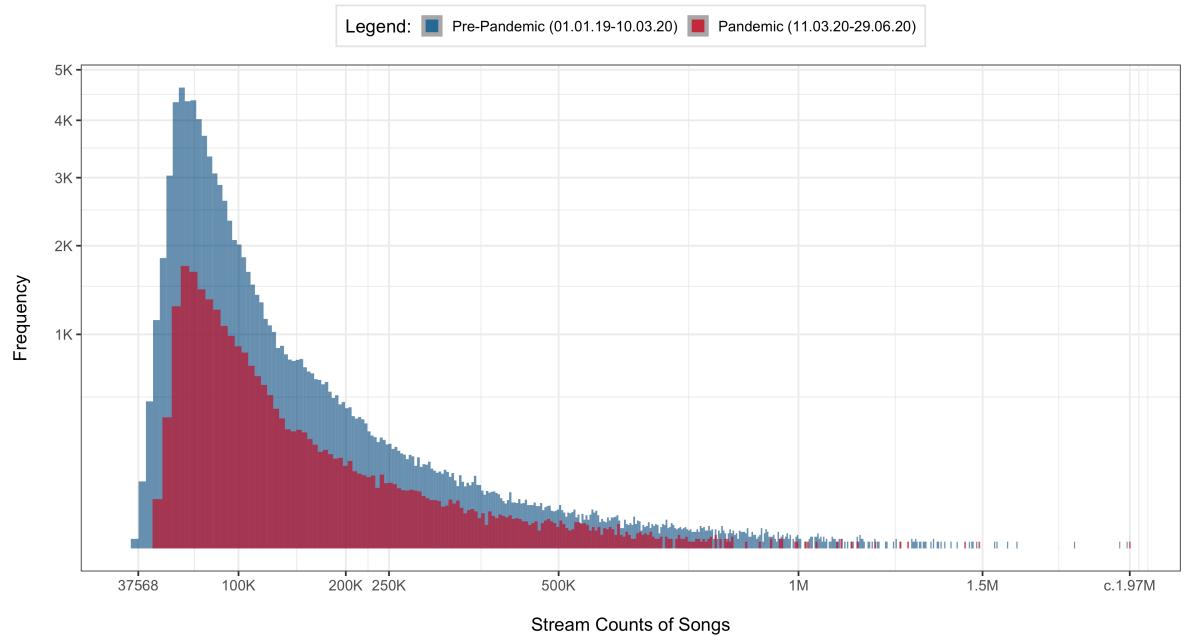

Figure 1: Histogram: Stream counts of Germany's daily top 200 Spotfy charts (square root-scaling).

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 were streamed between almost 40,000 and 500,000 times, whereas only few songs and tracks were 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 daily stream counts.

2.1 Analyzing Differences in Daily Stream Counts Before and During the First Wave of the COVID-19 Pandemic

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 during the first wave of 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 split, 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:

```
# Splitting 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(date) %>%
                # Using the median instead of mean because of
                # skewed overall distribution (see histogram)
                summarize(Mdn_Streams = median(stream_count))

# Splitting 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(date) %>%
                     # Using the median instead of mean because of
```

```
                    # skewed overall distribution (see histogram)
                    summarize(Mdn_Streams = median(stream_count))
```

The reason for splitting 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 was reported according to the daily report of the Robert Koch Institute (2020). As the nationwide physical distancing orders were 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 split 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 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 one-way Wilcoxon sign-rank-test or its powerful parametric analogon, the t-test, are being up to choose. To decide which test is the most appropriate one, we should check whether the assumptions of the t-test are met (i.e., normal distributed data and variance homogeneity). For if these conditions are not met, then the non-parametric version, the Wilcoxon test, needs to be conducted.

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

# 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 are 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 do not go in-line with the assumptions of the t-test: The differences of the stream counts between those two periods are significantly different from a normal distribution according to the Shapiro-Wilk-Test for normality ($W = .939, p < .001$), so we have to 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 ($F_{(110,110)} = .779$, $p = .191$, $CL = 95\%$) which is why we can assume that both variances are essentially the same. Hence, we can now finally test for a potential significant difference by performing a paired

one-way Wilcoxon 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. But before we examine these assumptions, it might be helpful to recall very briefly the rationale of the Wilcoxon test.

Depending on whether differences of central tendencies of two dependent (matching) or independent sets are being up to test, different versions of the Wilcoxon need to be performed (for independent samples the `wilcox.test` function computes in fact the Mann-Withney-U test). Since we are here confronted with two depending sets (i.e., we compare the same period/range: the pre-pandemic period and the pandemic), we need to calculate a Wilcoxon test for dependent sets/samples.

Since a computed p-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 (regarding the rationale of effect sizes see Cohen, 1992), where $r = 0$ means that the significant result of the Wilcoxon 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 z-value and the sample size (n).

```
# Null hypothesis: No significant difference is detectable. That is, we "want"
# here a p-value equal to or less 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 stream counts during the pandemic are higher
# (i.e., "greater") than during the pre-pandemic period.

res <- wilcox.test( spotifyR_pandemic$Mdn_Streams, spotifyR_pre$Mdn_Streams,
                    paired = T, exact = F, alternative = "g",
                    conf.level = .95)
print(res)

# calculating the z score
z_value <- qnorm(res$p.value)
print(z_value)

# calculating effect size
effsize <- abs(z_value)/sqrt(111)
print(effsize)

# getting the summary statistics for both periods:

# Pre-Pandemic
spotifyR_pre %>%
  get_summary_stats(Mdn_Streams, type = "median_iqr")

# Pandemic
spotifyR_pandemic %>%
  get_summary_stats(Mdn_Streams, type = "median_iqr")
```

So, after carrying out the paired one-way Wilcoxon test and calculating the effect size of the observed significant result, we can state a moderate 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 ($p < .001$, $z = -3.43$, $r = .325$, $CL = 95\%$, $n = 111$). Resulting from this, we can now conclude that the average of the median stream counts of Germany's daily top 200 Spotify charts during the pandemic-period is significantly higher with a moderate effect of difference compared to the pre-pandemic period ($Mdn_{Pre-Pandemic} = 89215.5$ and $IQR_{Pre-Pandemic} = 15820.5$; $Mdn_{Pandemic} = 94545.5$ and $IQR_{Pandemic} = 10318.2$). We can see this moderate effect of difference in the following graph:

```
# 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")

Mdn_df <- tibble(Date = c(as.Date(spotifyR_pre$date),
                          as.Date(spotifyR_pandemic$date)),
          Mdn_Stream_Counts = c(spotifyR_pre$Mdn_Streams,
                                spotifyR_pandemic$Mdn_Streams))

Mdn_df$Period <- ifelse(Mdn_df$Date < as.Date("2020-03-11"),
                        "Pre-Pandemic (11.03.19-29.06.19)",
                        "Pandemic (11.03.20-29.06.20)")

ggplot(data = Mdn_df) +

# Connected Scatter plot
geom_point(aes(Date, Mdn_Stream_Counts, color = Period), show.legend = F) +

geom_line(aes(Date, Mdn_Stream_Counts, group = 1, color = Period),
          size = .75, show.legend = F) +

facet_wrap(.~factor(Period, levels =c("Pre-Pandemic (11.03.19-29.06.19)",
                                      "Pandemic (11.03.20-29.06.20)")),
           scales = "free") +

# 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 = "\nDate of Streams", y = "Stream Counts\n(Median)") +

# Label numbers with SI prefixes
scale_y_continuous(labels = label_number_si(accuracy = NULL)) +
scale_x_date(date_breaks = "1 month")+

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

# Layout
theme_bw(base_size = 12) +
theme(axis.title.y = element_text(size = 12),
      axis.title.x = element_text(size = 12),
      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(.99,.66,.99,.66), "cm"))

# Saving the output (wide format)
ggsave("weeks_scatter.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
        width = 12.8, height = 7.2, dpi = 320)
```

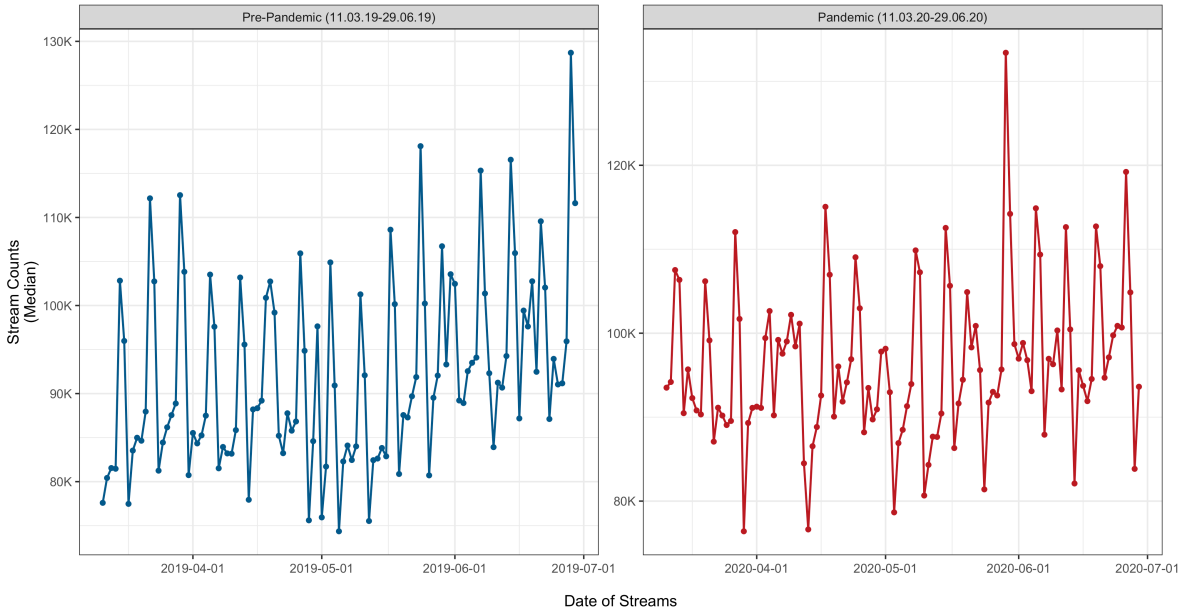

Figure 2: Connected Scatter Plot: Stream Counts (Median) of Germany's Daily Top 200 Spotify Charts Before and During the Pandemic.

That is, we have here no reason to accept and, respectively, retain the null hypothesis of the Wilcoxon test, whose assumption purports that *no* significant differences in the rank sums of the compared periods are detectable at all.

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 Toward a Prediction Model

As we have detected a significant difference between those two periods after performing a Wilcoxon test and estimated a moderate 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 January 1st, 2019, and June 29th, 2020, we should create once again a connected scatter plot like that one we have already done it for the daily stream counts data of both periods, 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 stream 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)+

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 = "3 month", date_labels = "%Y-%m")+
# Label numbers with SI prefixes
scale_y_continuous(labels = label_number_si(accuracy = NULL))+
# 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),
      plot.margin = unit(c(.66, .33, .33, .33), "cm"))

# Saving the output (wide format)
ggsave("p2.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
       width = 12.8, height = 7.2, dpi = 320)
```

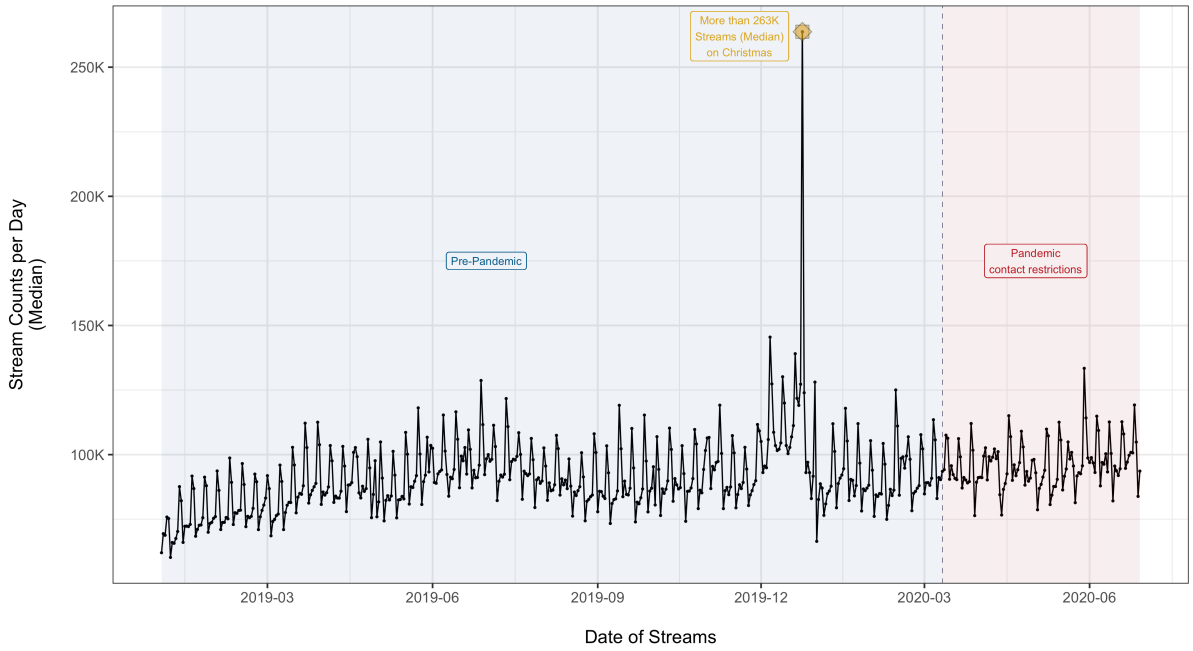

Figure 3: Connetected Scatter Plot: Daily Stream Counts of Germany's Top 200 Spotify-Charts (January 1st, 2019 to June 29th, 2020).

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 a 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 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 the 10th of March needs to be
# plugged-in as the day number of the year 2020. To identify this day number, we
# can simply use the yday-function of the lubridate package:

# Using the ISO-date format to identify the day number of the year 2020
```

```
print(yday("2020-03-10"))

# The 10th of March, 2020, is day 70 of the whole 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/define the time period that we want to predict (i.e., the first wave of the pandemic).As the pandemic period consists of 111 days, as we know from the difference test in chapter 2 of this report, the entire pre-pandemic period consists of 435 days, which equals to c 80% of training data for the prediction task. Accordingly, the 111 days–as the test set in question–are equal to c.20% of the whole data. Hence, the split ratio for the prediction task in question is about 80 to 20.

```
# Day difference between March 11th, 2020, and June 29th, 2020: Since we aim to
# estimate the median stream counts of these days (that is the analyzed
# period of the COVID-19-pandemic of interest including start and end days), we
# need to add 1 to the difference which we get when using lubridate's
# "as.duration" and "ddays"-functions.

pred_period <- as.duration("2020-03-11" %--% "2020-06-29") / ddays(1) + 1
print(pred_period)
```

Since we get predictions of the actual median stream counts with our trained models later on, we should compare these estimates with a null 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 prediction period (i.e., March 11th, 2020, to June 29th, 2020) are equal to the values of the same period of the last observed season (i.e., in our case March 11th, 2019, to June 29th, 2019).

```
# Seasonal-naïve predictions
null_model <- snaive(spotifyR_TS_orig, h = pred_period)

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

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

So with this, we have a null model that should be outperformed with our models in question. In particular, we should achieve with our models lesser errors 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 null 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 = "14 day", date_labels = "%Y-%m-%d") +

# 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 = 12.8, height = 7.2, dpi = 320)
```

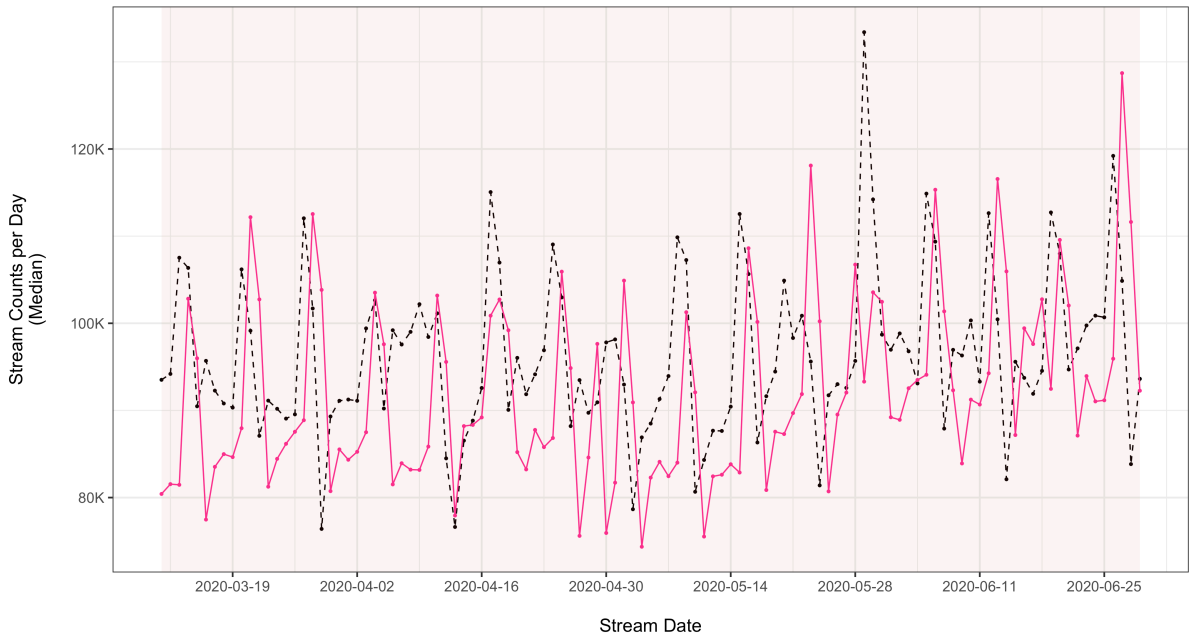

Figure 4: Connected Scatter Plot: Comparison of Actual Stream Counts and Seasonal-Naïve Fits.

Now we should examine whether the original time series follows a trend and whether the data consist of any seasonal pattern. Based on the plots from above, we can assume that the original time series is not yet stationary since increasing weekly patterns are observable. Accordingly, we should difference the data so the predictions rely not simply on the actual values but rather on their differences–with this, we can finally control seasonal and trend effects.

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

diffs <- ndiffs(spotifyR_TS_orig)
```

```
print(diffs)

# 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 = diffs)
```

Based on this transformation we can reasonably assume that our time series is indeed stationary and does not imply a trend. This can be vizualized as follows:

```
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 = 12.8, height = 7.2, dpi = 320)
```

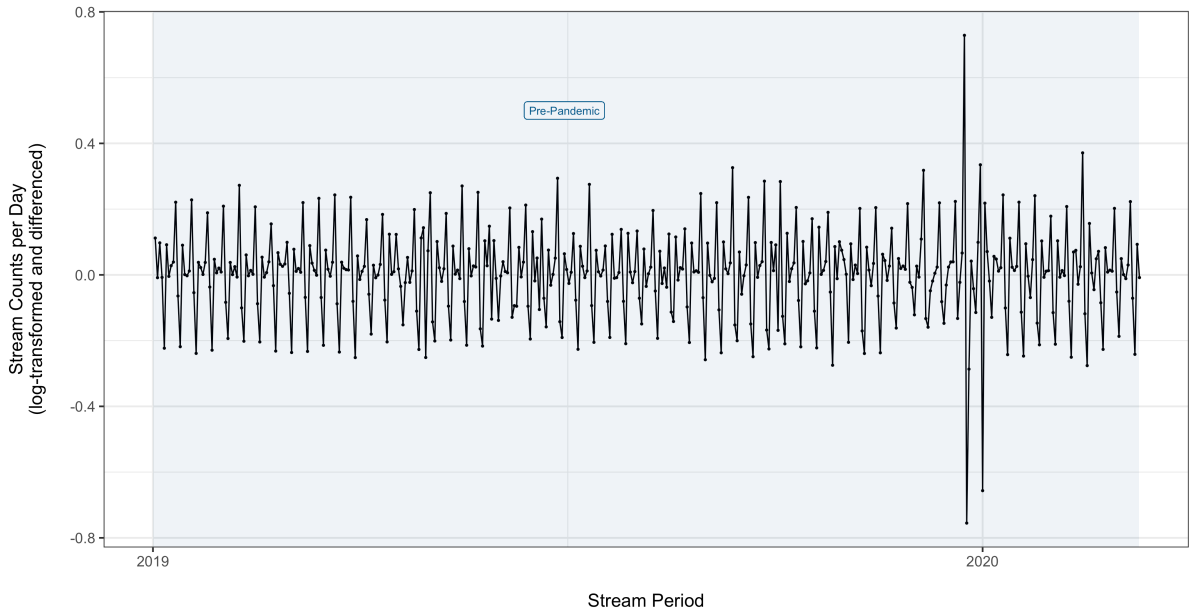

Figure 5: Stationarized pre-pandemic Period.

To test this visual impression statistically, we can run two tests with opposite null hypotheses to be sure that we are indeed dealing with a stationary time series.

For doing so, we will test whether the time series follows a trend (time-dependency) by using the so-called Augmented Dickey Fuller (ADF) test whose null hypothesis states that the time series is non-stationary (see for a brief introduction Holmes et al., 2020). On the other hand, to test whether the time series is stationary (opposite null hypothesis), we will conduct the so-called KPSS test (see Holmes et al., 2020). Hence, if we can reject the null hypothesis of the ADF test and fail to reject the null hypothesis of the KPSS test, we can assume that the time series in question is indeed stationary.

```
# ADF: Null hypo: Time series is non-stationary. That is, we "want" reject the
# null (i.e., p < .05, an absolute test statistic greater than the critical
# value at an alpha-level of 5%))

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

# KPSS: Null hypo: Time series is stationary. Here, we "want" to fail to reject
# the null hypo (i.e., a test statistic less than the critical value at
# an alpha-level of 5%)

spotifyR_TS_trans %>%
# we choose coefficient "mu" as the constant since we already have transformed
# the data (we also do not specify the number of lags at this point, as we will
# determine the optimal lag number in the next step)
ur.kpss(type = "mu", lags = "short") %>%
summary()
```

Now, both test results support the assumption of a stationary time series ($ADF_{\tau 1} = |25.9| > |1.95|$, $CL = 95\%$; $KPSS_{\mu} = 0.0312 < 0.463$, $CL = 95\%$). However, as we also see a repeating pattern in the differenced time series, its likely that the time series in question contains mutual information about its different time steps (i.e., the repeating pattern). If so, we need to take care of this issue before we build our model (see e.g., Wallot & Mønster, 2018).

```r
# How many lags (i.e., repeating frequency pattern) does the period consist?

lag_selection <- mutualInformation(spotifyR_TS_trans, do.plot = F,
                                   # natural logs
                                   units = "Nats")

print(lag_selection$n.partitions)
```

```
## [1] 7
```

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

lags <- 7
```

Before we begin to examine the second hypothesis H_2, we have to test whether the time series is autocorrelated. For doing so, we can simply plot the autocorrelation function against the identified lags with an included confidence band at a confidence level of 95%. That is, if the lags are within the confidence band, the time series is not to relevant degree autocorrelated.

```r
ggAcf(spotifyR_TS_trans, lag.max = 7)+
# Customizing the labels on both axes
labs(y = "ACF", x = "\nLags") +
ggtitle(label="")+
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)
# Saving the plot (wide format)
ggsave("autocorr.png", path = "~/Desktop/R_Projects/spotifyR/Plots",
       width = 12.8, height = 7.2, dpi = 320)
```

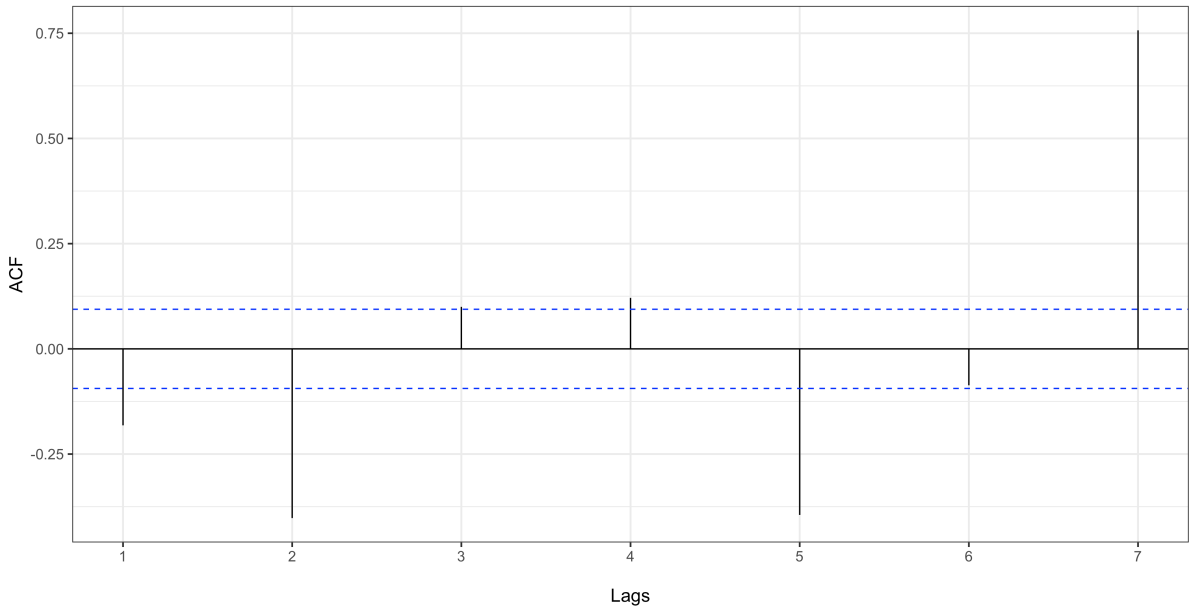

Figure 6: ACF Plot of the Pre-Pandemic Period (7 Lags).

As we can see, the time series is to some extent seriously autocorrelated (see lags 2, 5, and 7). This is an issue we have to take care of as the prediction accuracy would suffer. To overcome this issue, we can employ a technique that allows to out-cancel such errors.

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. As von Oertzen & Boker (2010) have demonstrated, such errors can be out-canceled for linear and non-linear problems. Following this approach, we transform the time series into a two-dimensional data structure (i.e., matrix). That is, we create a matrix where the suggested dimensions of lags (in our case 7) pose the predictors for the remaining index-data points. Thus, by embedding the time series into such a matrix, we get a eight column-matrix in total where the first column consists the original data points and the other seven columns consist of those stream 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., von Oertzen & Boker, 2010, Bontempi et al, 2013). This strategy refers originally to the so-called Tackens' Theorem, which is, by the way, also conceptually relevant for the dynamic systems theory within the scope of situated music cognition, see Demos et al., 2014).

```
# 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:  1 (variable to be predicted) + 3 lags
# (predictors)

3 <- lags_example

>embed(x, 1 + lags_example)

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

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

```
spotifyR_TS_trans_embed <- embed(spotifyR_TS_trans, 1 + 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 k-Nearest Neighbors (kNN) algorithm, the second of which is the Random Forest 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 seasonal-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.

3.2 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 (c.20%). Thus, the pandemic period represents our already given test set, whereas the pre-pandemic period (c.80%) poses the training set.

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

y_train <- spotifyR_TS_trans_embed[,1]

# Regressor (aka: input, independent variables)

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; this is the
# examined period of the COVID-19 pandemic.
```

```r
# Using the ISO-date format to identify the day number of June 29th, 2020.
print(yday("2020-06-29"))

# June 29th, 2020, is the day 181 of the whole year.

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

As we now have split our data into training and test sets, we can begin to train our models.

4 Model Training

To begin with model training, 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 will run a five times repeated 10-fold cross-validations (to ensure robustness) to identify the optimal hyperparameters of the respective algorithms. This can be done with the following approach:

```r
# 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(1, 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])

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

# 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
# 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(321, 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])
```

Next, we can indeed train our models by using the cross-validated values for the respective hyperparameters.

4.1 Training Loops

Finally, we can now train our models with optimized hyperparameters by looping the models through every point of the prediction period (i.e., 111 days). That is, we essentially compute 111 models: One model for each data point of the prediction period (regarding the rationale of this strategy see, e.g., Bontempi et al., 2013).

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

knn_estimates <- numeric(pred_period)

# For-loop: kNN Training

for (i in 1:pred_period) {

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

# Here we fill the mentioned "container" with the predicted values

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

# The training data is here constantly updated to account for the time distance
# corresponding to the respective data point of the prediction period. That is,
# once one data point is predicted, it will be accordingly subtracted.

# Subtracting/Removing the respective first data point.
y_train <- y_train[-1]

# Subtracting/Removing the respective last data point.
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 rf-algorithm

rf_estimates <- numeric(pred_period)

# For-loop: Random Forest Training

for (j in 1:pred_period) {
```

```r
set.seed(0, sample.kind = "Rounding")

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

# Here we fill the mentioned "container" with the predicted values

rf_estimates[j] <- 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 prediction period. That is,
# once one data point is predicted, it will be accordingly subtracted.

y_train <- y_train[-1]

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

Having now trained our models, we need to convert the predicted values to their former scale.

4.2 Converting Estimates

Since we transformed the original data, we have to convert them to get meaningful/interpretable values. For doing so, we create a function that takes the median values (as a robust reference value, alternatively the last value of the training set would also be appropriate but not as robust as the median) of the original training data and multiplies it with the running totals (i.e., the cumulative sums) of the transformed estimates to reverse our differencing procedure. Then, we exponentiate each of these cumulative sums to reverse the previous logarithms.

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

converter <- function(x,y){# taking the median value of the original training data
                           # to multiply it with the values of the exponents of
                           # the cumulative sums of the predicted values.
                            value <- c(median(x) * exp(cumsum(y)))
                            }

# Getting the final values by using the median observation of the original
# training data as the reference value to which the predicted
# difference refers.

knn_conv_estimates <- converter(spotifyR_TS_orig, knn_estimates)
rf_conv_estimates <- converter(spotifyR_TS_orig, rf_estimates)

# Converting them into time series-format

knn_y_pred <- ts(knn_conv_estimates, start = c(2020, 71),
                 frequency = 365)

rf_y_pred <- ts(rf_conv_estimates, start = c(2020, 71),
                frequency = 365)
```

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

5 Evaluating Test Results

Now, we will see the error metrics. This is a crucial part with regard to our hypothesis. For if the error values are 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(SN_Estimates = c(rep(NA, length(spotifyR_TS_orig)), null_model$mean),
       kNN_Estimates = c(rep(NA, length(spotifyR_TS_orig)), knn_y_pred),
       rf_Estimates = c(rep(NA, length(spotifyR_TS_orig)), rf_y_pred))

# Extracting only the pandemic-period

test <- tail(spotifyR_tib_star, pred_period)

# Calculating the error metrics
fuzzyr.accuracy(test$kNN_Estimates, test$Actual_Stream_Counts, test$SN_Estimates)
fuzzyr.accuracy(test$rf_Estimates, test$Actual_Stream_Counts, test$SN_Estimates)
```

When evaluating the results ($RMSE_{kNN}$ = 6400.86, $MAPE_{kNN}$ = 5.05, $RMSE_{RF}$ = 6808, $MAPE_{RF}$ = 5.57), we see that both models perform quite well while the random forest model yields slightly more errors. Accordingly, we can state our models performs a way better (c.53.2% higher accuracy) than the seasonal-naïve method ($UMBRAE_{kNN:SN}$ = 0.468; for the UMBRAE accuracy metric see Chen et al., 2017). This is insofar satisfying as our approach indeed could outperform the null model. But as usual, we should 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 kNN model) are within the prediction interval, we should first calculate these prediction intervals (see Hyndman & Athanasopoulos, 2018).

```
# Checking whether the residuals come from a normal distribution.
resids <- test$Actual_Stream_Counts - test$kNN_Estimates
shapiro.test(resids)

# Since the residuals are closely sufficient normally distributed according to
# the Shapiro-Wilk- Test (W = 0.9784, p = 0.0689) based on an alpha-level of 5 %,
# 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. Alternatively, we could also
# bootstrap the predictions intervals.

SD_star <- sd(resids)

spotifyR_tib_star %<>%
mutate(kNN_E_low_95 = kNN_Estimates - (1.96 * SD_star),
       kNN_E_high_95 = kNN_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 kNN model.

```
# Creating a color-object

cols <- c("Actual Stream Counts (Median)" = "black",
          # Colorblind Green
          "Random Forest Predictions" = "darkgoldenrod2",
          # Colorblind Orange
          "kNN Predictions\nwithin 95% PI" = "darkblue",
          # Colorblind Pink
          "Seasonal-naïve Predictions" = "deeppink")

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 = kNN_Estimates, color = "kNN Predictions\nwithin 95% PI"),
           size = .75) +
geom_line(aes(y = kNN_Estimates, color = "kNN Predictions\nwithin 95% PI"),
          lty = "solid") +

geom_point(aes(y = rf_Estimates,
               color = "Random Forest Predictions"),
              size = .75) +
geom_line(aes(y = rf_Estimates,
               color = "Random Forest 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 = kNN_Estimates, ymin = kNN_E_low_95, ymax = kNN_E_high_95,
  fill = "kNN Predictions\nwithin 95% PI"), alpha = 0.2, show.legend = F) +

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 = 132e3,
  label = "Test Dataset\n(Pandemic Period)"),  color = "darkred",
  size = 3) +
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 = "14 day", date_labels = "%Y-%m-%d") +

# 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(.99, .33, .99, .33), "cm"))
# Saving the output (wide format)
ggsave("p6.png", path = "~/Desktop/R_Projects/spotifyR/Plots", device = "png",
       width = 12.8, height = 7.2, dpi = 320)
```

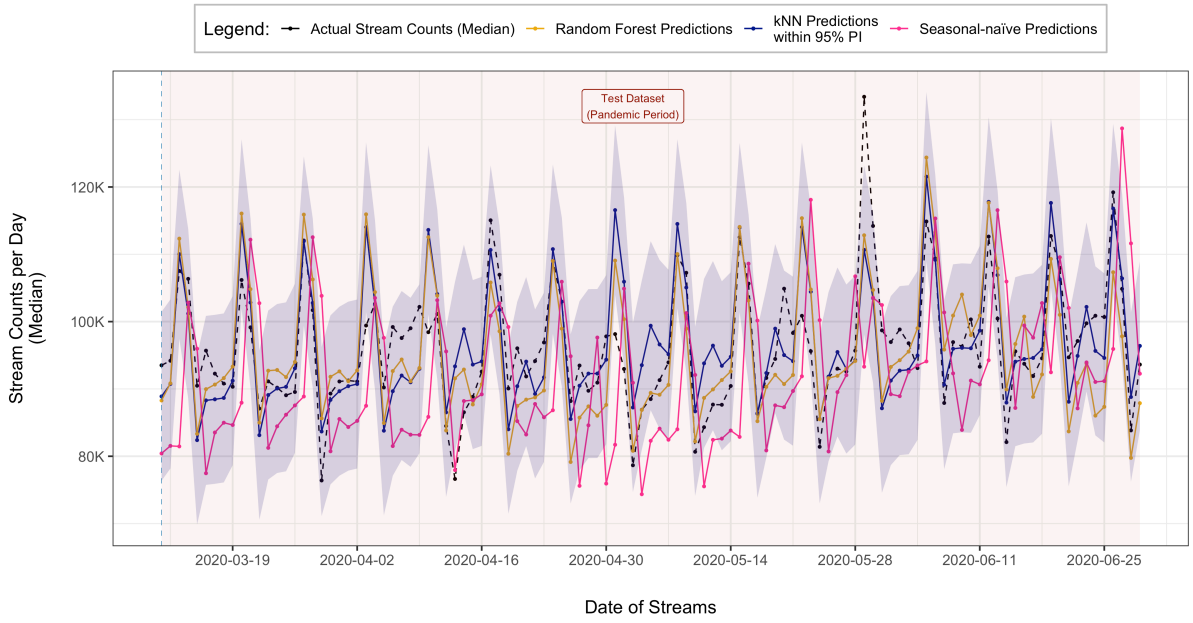

Figure 7: Comparison of Actual Stream Counts, kNN, Random Forest Estimates, and Seasonal-Naïve Fits.

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. All in all, we can state that the actual stream counts and our kNN-estimates go along–with some deviations (see for example, May 29th).

Conclusion

At the ending of our endeavor, we can answer our initial question with the tested hypothesis now as follows: Germans tend to listen to more music during the COVID-19 pandemic. At least, there are significant and strong differences within a week interval between the pre-pandemic and the pandemic periods observable. Hence, we do not have evidence to reject our first hypothesis. Moreover, we could also predict the median stream counts of the pandemic period (first wave) based on dates with desirable error values by using Random Forest and kNN-regressions.

In this vein, it would be also interesting to compare the performance of our final model (kNN model) with even more advanced tree-based models, such as Extreme Gradient Boosting, or with classic time series forecasting models. Possible next steps could be to run a sentiment analysis on the selected songs and tracks and/or to examine to what extent such sentiments are related with audio features of the respective tracks that can be retrieved by using Spotify's API. On the other hand, it would also be interesting to run this model on stream count data of daily top 200 charts during the current period of the COVID-19 pandemic and/or of the daily top 200 charts of other German-speaking countries (i.e., Austria and Switzerland) to examine and to optimize the performance of our model. Besides that, our demonstrated modeling strategy might be also useful when it comes to date sensitive music recommender systems.

Above all, this approach illustrates—despite its conceptual limitations when it comes to examining music streaming behavior not only via such proxy variables—how null hypothesis significance tests, web-scraping, and supervised machine learning techniques can be employed to predict daily music-listening

behavior at least via a proxy variable (i.e., stream counts) by using open data.

Appendix – Environment

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

```
## Loading required package: sessioninfo
```

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 4.1.0 (2021-05-18)
##  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     2021-07-28
##
## - Packages -------------------------------------------------------------------
##  package          * version    date       lib source
##  abind              1.4-5      2016-07-21 [1] CRAN (R 4.1.0)
##  anytime            0.3.9      2020-08-27 [1] CRAN (R 4.1.0)
##  assertthat         0.2.1      2019-03-21 [1] CRAN (R 4.1.0)
##  backports          1.2.1      2020-12-09 [1] CRAN (R 4.1.0)
##  broom              0.7.6      2021-04-05 [1] CRAN (R 4.1.0)
##  car                3.0-10     2020-09-29 [1] CRAN (R 4.1.0)
##  carData            3.0-4      2020-05-22 [1] CRAN (R 4.1.0)
##  caret            * 6.0-88     2021-05-15 [1] CRAN (R 4.1.0)
##  cellranger         1.1.0      2016-07-27 [1] CRAN (R 4.1.0)
##  class              7.3-19     2021-05-03 [1] CRAN (R 4.1.0)
##  cli                3.0.0      2021-06-30 [1] CRAN (R 4.1.0)
##  codetools          0.2-18     2020-11-04 [1] CRAN (R 4.1.0)
##  colorspace         2.0-2      2021-06-24 [1] CRAN (R 4.1.0)
##  crayon             1.4.1      2021-02-08 [1] CRAN (R 4.1.0)
##  curl               4.3.2      2021-06-23 [1] CRAN (R 4.1.0)
##  data.table         1.14.0     2021-02-21 [1] CRAN (R 4.1.0)
##  DBI                1.1.1      2021-01-15 [1] CRAN (R 4.1.0)
##  dbplyr             2.1.1      2021-04-06 [1] CRAN (R 4.1.0)
##  digest             0.6.27     2020-10-24 [1] CRAN (R 4.1.0)
##  dplyr            * 1.0.7      2021-06-18 [1] CRAN (R 4.1.0)
##  e1071              1.7-7      2021-05-23 [1] CRAN (R 4.1.0)
##  ellipsis           0.3.2      2021-04-29 [1] CRAN (R 4.1.0)
##  evaluate           0.14       2019-05-28 [1] CRAN (R 4.1.0)
##  expsmooth        * 2.3        2015-04-09 [1] CRAN (R 4.1.0)
##  fansi              0.5.0      2021-05-25 [1] CRAN (R 4.1.0)
##  fastmap            1.1.0      2021-01-25 [1] CRAN (R 4.1.0)
##  fma              * 2.4        2020-01-14 [1] CRAN (R 4.1.0)
##  FNN              * 1.1.3      2019-02-15 [1] CRAN (R 4.1.0)
##  forcats          * 0.5.1      2021-01-27 [1] CRAN (R 4.1.0)
##  foreach            1.5.1      2020-10-15 [1] CRAN (R 4.1.0)
##  forecast         * 8.15       2021-06-01 [1] CRAN (R 4.1.0)
##  foreign            0.8-81     2020-12-22 [1] CRAN (R 4.1.0)
##  fpp2             * 2.4        2020-09-09 [1] CRAN (R 4.1.0)
##  fracdiff           1.5-1      2020-01-24 [1] CRAN (R 4.1.0)
##  fs                 1.5.0      2020-07-31 [1] CRAN (R 4.1.0)
##  FuzzyR           * 2.3.2      2021-05-19 [1] CRAN (R 4.1.0)
##  generics           0.1.0      2020-10-31 [1] CRAN (R 4.1.0)
```

```
##  ggExtra          * 0.9        2019-08-27 [1] CRAN (R 4.1.0)
##  ggplot2          * 3.3.5      2021-06-25 [1] CRAN (R 4.1.0)
##  ggrepel          * 0.9.1      2021-01-15 [1] CRAN (R 4.1.0)
##  glue               1.4.2      2020-08-27 [1] CRAN (R 4.1.0)
##  gower              0.2.2      2020-06-23 [1] CRAN (R 4.1.0)
##  gtable             0.3.0      2019-03-25 [1] CRAN (R 4.1.0)
##  haven              2.4.1      2021-04-23 [1] CRAN (R 4.1.0)
##  hms                1.1.0      2021-05-17 [1] CRAN (R 4.1.0)
##  htmltools          0.5.1.1    2021-01-22 [1] CRAN (R 4.1.0)
##  httpuv             1.6.1      2021-05-07 [1] CRAN (R 4.1.0)
##  httr               1.4.2      2020-07-20 [1] CRAN (R 4.1.0)
##  ipred              0.9-11     2021-03-12 [1] CRAN (R 4.1.0)
##  iterators          1.0.13     2020-10-15 [1] CRAN (R 4.1.0)
##  jsonlite           1.7.2      2020-12-09 [1] CRAN (R 4.1.0)
##  kableExtra       * 1.3.4      2021-02-20 [1] CRAN (R 4.1.0)
##  knitr            * 1.33       2021-04-24 [1] CRAN (R 4.1.0)
##  later              1.2.0      2021-04-23 [1] CRAN (R 4.1.0)
##  lattice          * 0.20-44    2021-05-02 [1] CRAN (R 4.1.0)
##  lava               1.6.9      2021-03-11 [1] CRAN (R 4.1.0)
##  lifecycle          1.0.0      2021-02-15 [1] CRAN (R 4.1.0)
##  lmtest           * 0.9-38     2020-09-09 [1] CRAN (R 4.1.0)
##  lubridate        * 1.7.10     2021-02-26 [1] CRAN (R 4.1.0)
##  magrittr         * 2.0.1      2020-11-17 [1] CRAN (R 4.1.0)
##  MASS             * 7.3-54     2021-05-03 [1] CRAN (R 4.1.0)
##  Matrix             1.3-3      2021-05-04 [1] CRAN (R 4.1.0)
##  mime               0.11       2021-06-23 [1] CRAN (R 4.1.0)
##  miniUI             0.1.1.1    2018-05-18 [1] CRAN (R 4.1.0)
##  ModelMetrics       1.2.2.2    2020-03-17 [1] CRAN (R 4.1.0)
##  modelr             0.1.8      2020-05-19 [1] CRAN (R 4.1.0)
##  munsell            0.5.0      2018-06-12 [1] CRAN (R 4.1.0)
##  nlme               3.1-152    2021-02-04 [1] CRAN (R 4.1.0)
##  nnet               7.3-16     2021-05-03 [1] CRAN (R 4.1.0)
##  nonlinearTseries * 0.2.11     2021-05-12 [1] CRAN (R 4.1.0)
##  openxlsx           4.2.3      2020-10-27 [1] CRAN (R 4.1.0)
##  pillar             1.6.1      2021-05-16 [1] CRAN (R 4.1.0)
##  pkgconfig          2.0.3      2019-09-22 [1] CRAN (R 4.1.0)
##  plyr               1.8.6      2020-03-03 [1] CRAN (R 4.1.0)
##  pROC               1.17.0.1   2021-01-13 [1] CRAN (R 4.1.0)
##  prodlim            2019.11.13 2019-11-17 [1] CRAN (R 4.1.0)
##  promises           1.2.0.1    2021-02-11 [1] CRAN (R 4.1.0)
##  proxy              0.4-26     2021-06-07 [1] CRAN (R 4.1.0)
##  purrr            * 0.3.4      2020-04-17 [1] CRAN (R 4.1.0)
##  quadprog           1.5-8      2019-11-20 [1] CRAN (R 4.1.0)
##  quantmod           0.4.18     2020-12-09 [1] CRAN (R 4.1.0)
##  R6                 2.5.0      2020-10-28 [1] CRAN (R 4.1.0)
##  randomForest     * 4.6-14     2018-03-25 [1] CRAN (R 4.1.0)
##  ranger           * 0.12.1     2020-01-10 [1] CRAN (R 4.1.0)
##  Rcpp               1.0.6      2021-01-15 [1] CRAN (R 4.1.0)
##  readr            * 1.4.0      2020-10-05 [1] CRAN (R 4.1.0)
##  readxl             1.3.1      2019-03-13 [1] CRAN (R 4.1.0)
##  recipes            0.1.16     2021-04-16 [1] CRAN (R 4.1.0)
##  reprex             2.0.0      2021-04-02 [1] CRAN (R 4.1.0)
##  reshape2           1.4.4      2020-04-09 [1] CRAN (R 4.1.0)
##  rio                0.5.26     2021-03-01 [1] CRAN (R 4.1.0)
##  rlang              0.4.11     2021-04-30 [1] CRAN (R 4.1.0)
##  rmarkdown          2.9        2021-06-15 [1] CRAN (R 4.1.0)
##  rpart              4.1-15     2019-04-12 [1] CRAN (R 4.1.0)
##  rstatix          * 0.7.0      2021-02-13 [1] CRAN (R 4.1.0)
```

```
##  rstudioapi         0.13       2020-11-12 [1] CRAN (R 4.1.0)
##  rvest            * 1.0.0      2021-03-09 [1] CRAN (R 4.1.0)
##  sandwich         * 3.0-1      2021-05-18 [1] CRAN (R 4.1.0)
##  scales           * 1.1.1      2020-05-11 [1] CRAN (R 4.1.0)
##  sessioninfo      * 1.1.1      2018-11-05 [1] CRAN (R 4.1.0)
##  shiny              1.6.0      2021-01-25 [1] CRAN (R 4.1.0)
##  stringi            1.7.3      2021-07-16 [1] CRAN (R 4.1.0)
##  stringr          * 1.4.0      2019-02-10 [1] CRAN (R 4.1.0)
##  strucchange      * 1.5-2      2019-10-12 [1] CRAN (R 4.1.0)
##  survival           3.2-11     2021-04-26 [1] CRAN (R 4.1.0)
##  svglite            2.0.0      2021-02-20 [1] CRAN (R 4.1.0)
##  systemfonts        1.0.2      2021-05-11 [1] CRAN (R 4.1.0)
##  tibble           * 3.1.2      2021-05-16 [1] CRAN (R 4.1.0)
##  tidyr            * 1.1.3      2021-03-03 [1] CRAN (R 4.1.0)
##  tidyselect         1.1.1      2021-04-30 [1] CRAN (R 4.1.0)
##  tidyverse        * 1.3.1      2021-04-15 [1] CRAN (R 4.1.0)
##  timeDate           3043.102   2018-02-21 [1] CRAN (R 4.1.0)
##  tseries            0.10-48    2020-12-04 [1] CRAN (R 4.1.0)
##  tsibble          * 1.0.1      2021-04-12 [1] CRAN (R 4.1.0)
##  TTR                0.24.2     2020-09-01 [1] CRAN (R 4.1.0)
##  urca             * 1.3-0      2016-09-06 [1] CRAN (R 4.1.0)
##  utf8               1.2.1      2021-03-12 [1] CRAN (R 4.1.0)
##  vars             * 1.5-3      2018-08-06 [1] CRAN (R 4.1.0)
##  vctrs              0.3.8      2021-04-29 [1] CRAN (R 4.1.0)
##  viridisLite        0.4.0      2021-04-13 [1] CRAN (R 4.1.0)
##  webshot            0.5.2      2019-11-22 [1] CRAN (R 4.1.0)
##  withr              2.4.2      2021-04-18 [1] CRAN (R 4.1.0)
##  xfun               0.24       2021-06-15 [1] CRAN (R 4.1.0)
##  xml2               1.3.2      2020-04-23 [1] CRAN (R 4.1.0)
##  xtable             1.8-4      2019-04-21 [1] CRAN (R 4.1.0)
##  xts                0.12.1     2020-09-09 [1] CRAN (R 4.1.0)
##  yaml               2.2.1      2020-02-01 [1] CRAN (R 4.1.0)
##  zip                2.2.0      2021-05-31 [1] CRAN (R 4.1.0)
##  zoo              * 1.8-9      2021-03-09 [1] CRAN (R 4.1.0)
##
## [1] /Library/Frameworks/R.framework/Versions/4.1/Resources/library
```