Initial Analysis of NHS Data using Python

Christian Putra Chen

22nd May 2023

London School of Economics and Political Science

Course: Data Analytics Career Accelerator
Module: Data Analytics using Python

1 Business Context and Background

The National Health Services (NHS) is a publicly-funded healthcare system in England that is trying to minimise the number of missed general practitioner (GP) appointments by adopting a data-focused approach. Each missed appointment incurs significant, potentially avoidable cost to the NHS, so a careful analysis of the data is required to establish ways to attack the root cause of the problem, rather than the surface-level symptoms. This project will attempt to unveil instructive insights hidden in current historic data on NHS records.

Key questions:

1. Has there been adequate staff and capacity in the networks?
2. What was the actual utilisation of resources?

2 Analytical Approach

When beginning any analytical project in **Python**, I adopt a systematic work flow in order to ensure that my code—housed in a **Jupyter Notebook**—is both insightful and easily understood. For data analysis specifically, the work flow is predictable and resembles the following:

1. Import all relevant libraries with best-practice aliases—in particular, **pandas**, **numpy**, **seaborn**, and **matplotlib.pyplot** (Figure A1).
2. Import the data sets as **pandas** DataFrames with clear variable names.
3. Sense check the data and clean if necessary (Figure A2).
4. If cleaning requires repetitive use of multi-line code, create a user-defined function to keep the code clean and speed up the process e.g. removing null values (Figure A3).
5. Upload the code to **GitHub** for version control.
6. Continue with the analysis, looping back to any previous steps if necessary.

It is also invaluable to write comments into the code as one progresses, so a detailed record of the analytical process is preserved.

To clean the relatively small data files provided, I would normally opt to use Excel for its ease of use and adaptability, but after importing **pandas**, converting said data files into DataFrames, then viewing the metadata in **Python** (Figure A4), these were my initial observations:

- **national_categories.xlsx**: The largest data set—"primary" file for initial analysis.
- **actual_duration.csv**: The next-largest data set—mostly technical information.
- **appointments_regional.csv**: A smaller data set—useful for later analysis.

None of the above data files had any null values (Figure A5), and the only cleaning that was required was changing the data types of certain columns as and when necessary. One integer-type column—**count_of_appointments**—was present in each of the three files, and this turned out to be an exceedingly useful quantity in all later analysis.

From this data, time-series visualisations were produced in order to explore possible seasonal trends in the number of appointments corresponding to different service settings (Figures A6 to A9). It was immediately clear that cyclic behaviour was present with a period of roughly seven days. The most likely explanation? The number of appointments vastly decreases on weekends; thus, in order to account for this cyclic nature, a simple moving average (SMA)—with a window size of seven days—was employed to make the seasonal trend more clear.

As the analysis progressed, it was eventually time to make use of the remaining data file containing tweet data from twitter, the initial observations being:

- **tweets.csv**: Small data set containing user tweets scraped from the twitter API.

I began analysis by wrangling the **#hashtags** from the **tweet_full_text** column through the use of user-defined functions (Figures A10 to A13); however, as the analysis progressed, I noticed that there was another column named **tweet_entities_hashtags** which contained pre-wrangled hashtags directly from the twitter API. After a thorough comparison and sense check, I decided that this data was more appropriate for the analysis and resultant visualisations (Figures A14 and A15).

3 Visualisation and Insights

Throughout this analysis, I generated many visualisations—some exploratory, and others explanatory. A select compendium of particularly interesting plots can be found in Section A of the appendix; however, in this section, I will focus strictly on plots which contribute the most crucial elements to resolving the business objectives posed.

We begin our journey into deeper analysis with a time series plot depicting the total number of appointments per month. As can be seen in Figure 1, the NHS accumulates a staggering 20M+ appointments every month, with occasional 30M+ peaks during busier months. Most of these appointments are attended to face to face, or over the telephone (Figure A16), by general practice staff under a GMS/PMS/APMS contract (Figure A6). The workload is roughly evenly split between GPs and other types of staff (Figure A17).

Seasonal trends also emerge. Figure 2 depicts a very clear drop off in number of appointments towards the end of autumn 2021, which bounces back up to baseline in winter, but is followed by sporadic drop offs throughout spring and summer. More analysis is required to determine

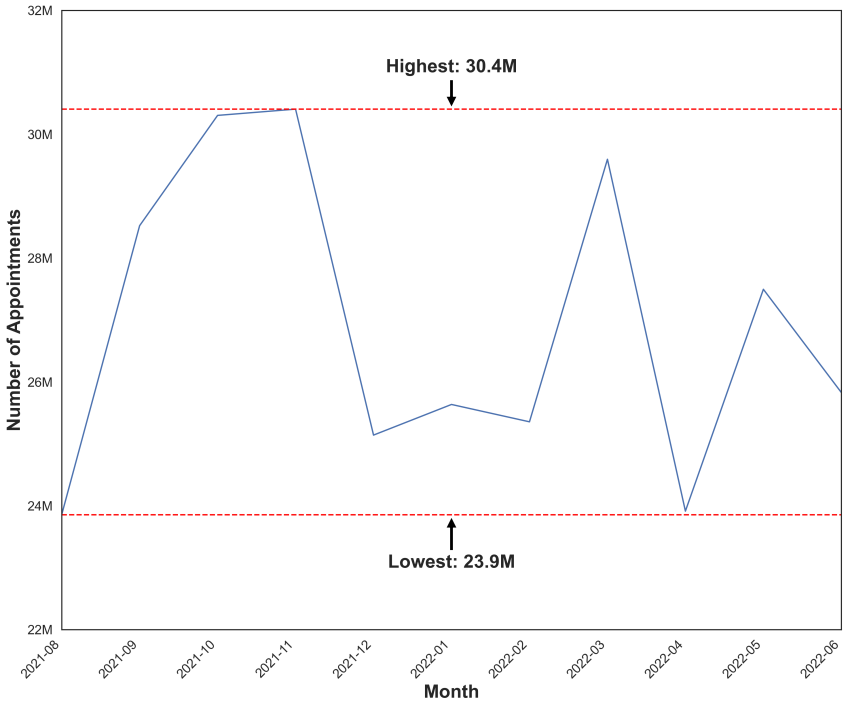

Figure 1: A time series of number of appointments per month, with maximum and minimum values indicated.

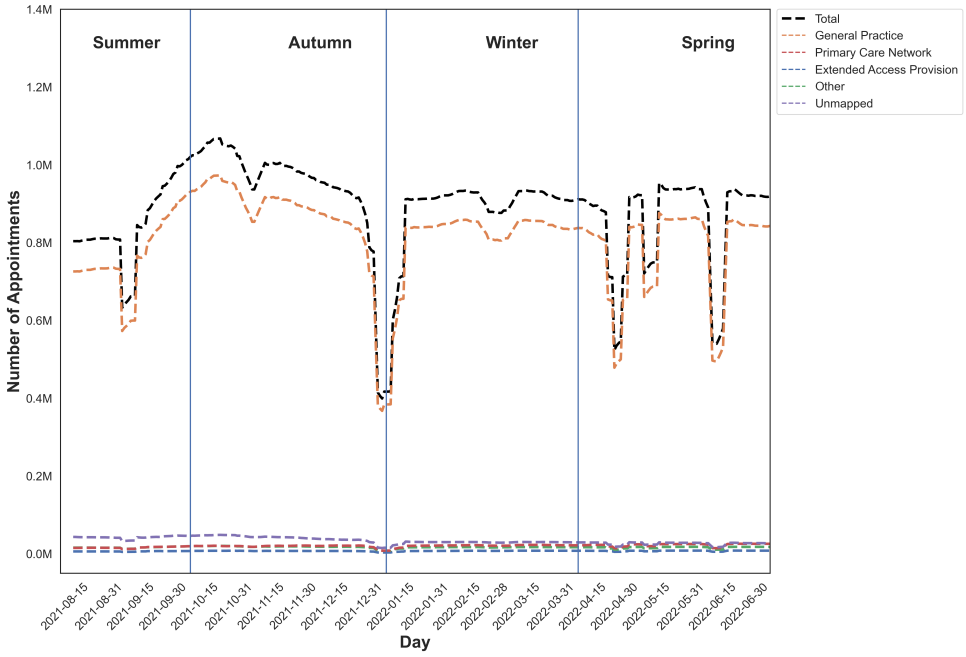

Figure 2: A time series of number of appointments per month, grouped by service setting, and aggregated using a simple moving average (SMA)—with a window of seven days—to remove noise from weekly fluctuations.

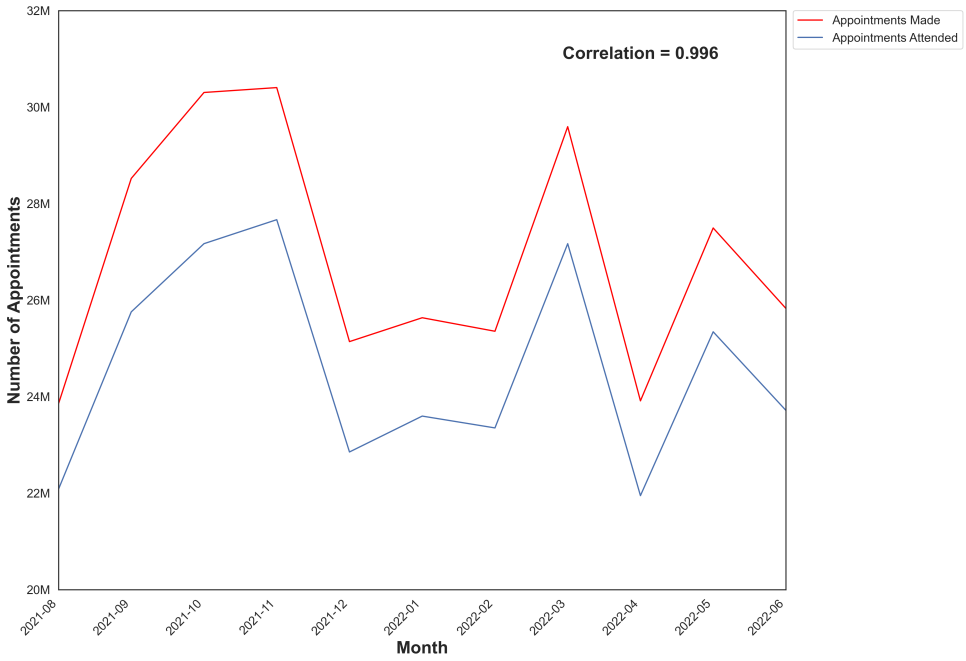

Figure 3: A time series of number of appointments per month—total, and also attended. The correlation between the two values is also indicated.

the exact nature of these sudden troughs, but a possible hypothesis for the autumn-winter drop off is increased staff sickness as winter approaches.

But how many people are attending these appointments? Figure 3 answers this question succinctly: the number of attended appointments correlates exceedingly well with the number of booked appointments (correlation = 0.996), thus indicating that appointment attendance rate

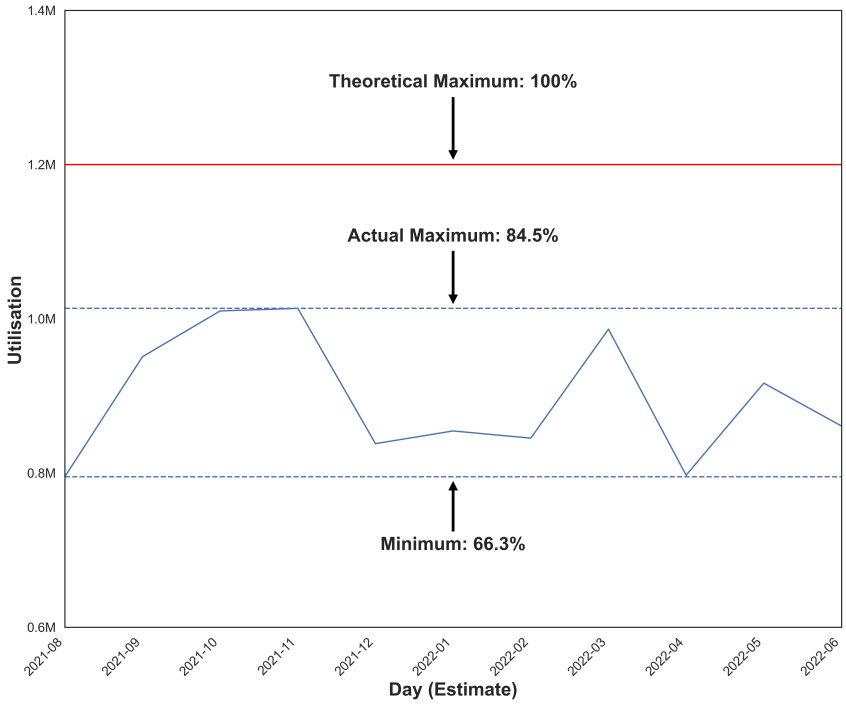

Figure 4: A time series of the estimated daily utilisation = (number of appointments per month)/30 compared to the theoretical maximum daily utilisation of 1.2M. Maximum and minimum utilisation is also indicated.

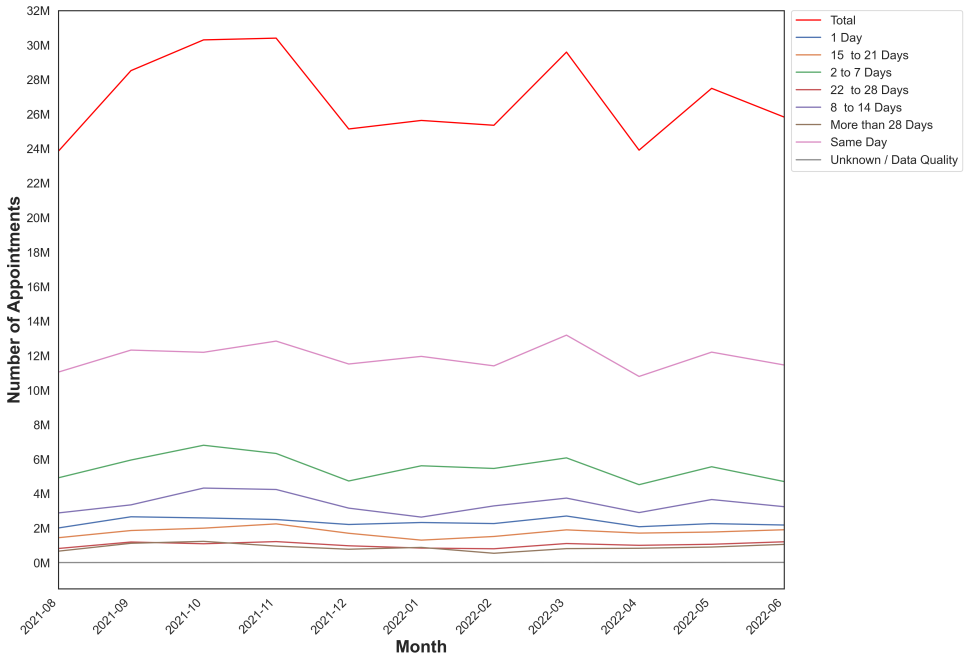

Figure 5: A time series of number of appointments per month, grouped by time between booking and appointment.

is governed by factors that remain consistent across time.

4 Patterns and Predictions

A striking insight can be gleaned from Figure 4, which depicts a calculated estimate of the NHS's daily utilisation. The theoretical maximum accommodation is 1.2M appointments per day (assuming 100% utilisation), but we can see that utilisation fluctuates between 66% and 85%, with the latter holding only for brief periods of time. The conclusion? Existing NHS staff are nowhere near fully utilised.

This is potentially due to systematic inefficiencies, trace evidence of which can be found in Figure 5. Here, we find that roughly half of appointments are attended to on the same day as the booking, and all others are delayed to various degrees. If perfect efficiency is defined as 100% of appointments being attended to on the same day, then the NHS is currently only 50% efficient, which strengthens the hypothesis that systematic inefficiencies are currently more of a problem than direct staff shortage.

Finally, according to the data, the following recommendations can be made:

- Addressing the underutilisation of current NHS resources should be the top priority—more staff will decrease wait times, but this is an expensive option that does not address the root problem of underlying systematic inefficiency.
- More analysis needs to be done to determine *why* patients miss appointments at a very predictable rate, independent of many external factors.
- It is imperative to make changes sequentially, and to keep track of patient attendance throughout, to track the success (or failure) of any particular systematic intervention.

Appendix A: Supplementary Figures

Import all relevant libraries and settings for analysis

In [2]:
```
# Import the necessary libraries for intitial analysis.
import pandas as pd
import numpy as np
from datetime import datetime
import random
import calendar
import string
```

In [3]:
```
# Import the necessary libraries for data visualisation.
import seaborn as sns
import matplotlib.pyplot as plt

# Set figure size.
sns.set(rc={'figure.figsize':(15, 12)})

# Set the plot style as white.
sns.set_style('white')
```

Figure A1: Importing libraries, with best-practice aliases, and adjusting relevant settings.

2.2 Importing and sense checking the data

In [5]:
```
# Import and sense-check the actual_duration.csv data. Name the DataFrame ad.
ad = pd.read_csv('actual_duration.csv')

# View the DataFrame by:
# 1. Checking the shape,
# 2. Checking the data types,
# 3. Checking the first five rows,
# 4. Checking the last five rows.
display(ad.shape)
display(ad.dtypes)
display(ad.head())
display(ad.tail())
```

Figure A2: Sense checking the data to determine if it was imported correctly and/or needs additional cleaning.

2.3 Checking for missing values

Functions:

In [8]:
```
# Build function that checks for missing values in DataFrame, and returns either:
# 1. The DataFrame containing only rows with null values,
# 2. Said DataFrame with null values highlighted in red (for visual checks).
def df_null(df, **kwargs):
    # Create a DataFrame subset containing missing values using isna().
    df_na = df[df.isna().any(axis=1)]

    # Provide optional keyword argument to highlight null values in red (if any exist).
    highlight = kwargs.get('highlight', None)

    # Return output DataFrame (or styler, if highlight=1)
    if highlight == 1:
        return df_na.style.highlight_null('red')
    else:
        return df_na
```

Figure A3: Creating a user-defined function that returns a DataFrame with only rows that contain at least one null value.

2.4 Reviewing the metadata.

```
In [13]: # Determine the metadata of data set actual_duration.csv.
         ad.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 137793 entries, 0 to 137792
Data columns (total 8 columns):
 #   Column                    Non-Null Count   Dtype 
---  ------                    --------------   ----- 
 0   sub_icb_location_code     137793 non-null  object
 1   sub_icb_location_ons_code 137793 non-null  object
 2   sub_icb_location_name     137793 non-null  object
 3   icb_ons_code              137793 non-null  object
 4   region_ons_code           137793 non-null  object
 5   appointment_date          137793 non-null  object
 6   actual_duration           137793 non-null  object
 7   count_of_appointments     137793 non-null  int64 
dtypes: int64(1), object(7)
memory usage: 8.4+ MB
```

Figure A4: Viewing the metadata using **pandas.DataFrame.info()**.

Outputs:

```
In [10]: # Determine whether there are missing values in actual_duration.csv.
         ad_na = df_null(ad)

         # Print the number of rows that have a NaN value.
         # Add docstring for clarity.
         print("The number of rows that have missing values:", ad_na.shape[0])

         # View the DataFrame containing only rows with null values.
         if ad_na.shape[0] != 0:
             display(ad_na)
```

```
The number of rows that have missing values: 0
```

Figure A5: Applying the **df_null()** function defined in Figure A3.

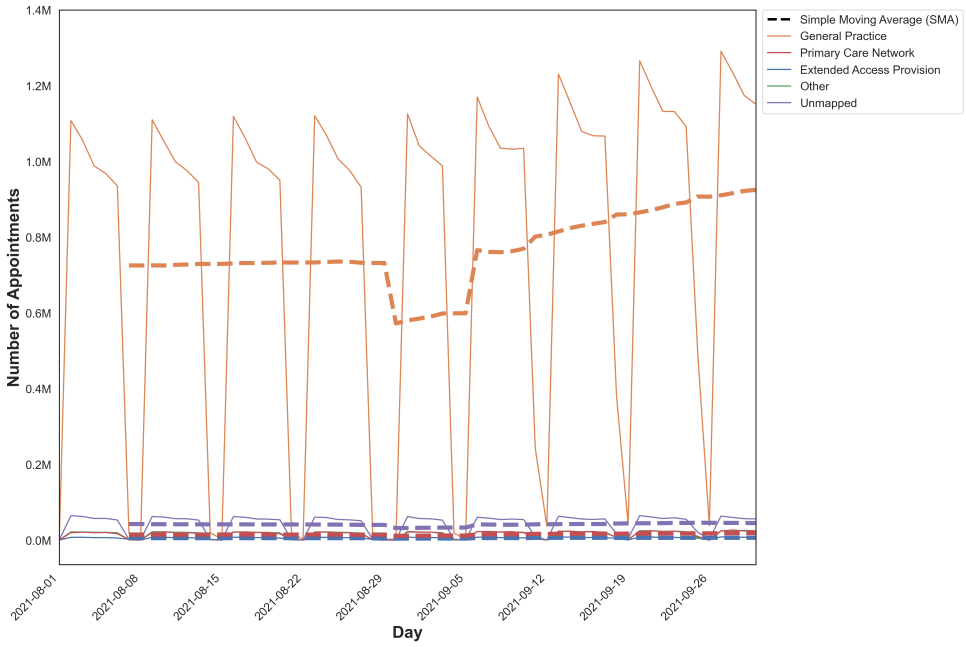

Figure A6: Time series of number of appointments per day throughout summer 2021.

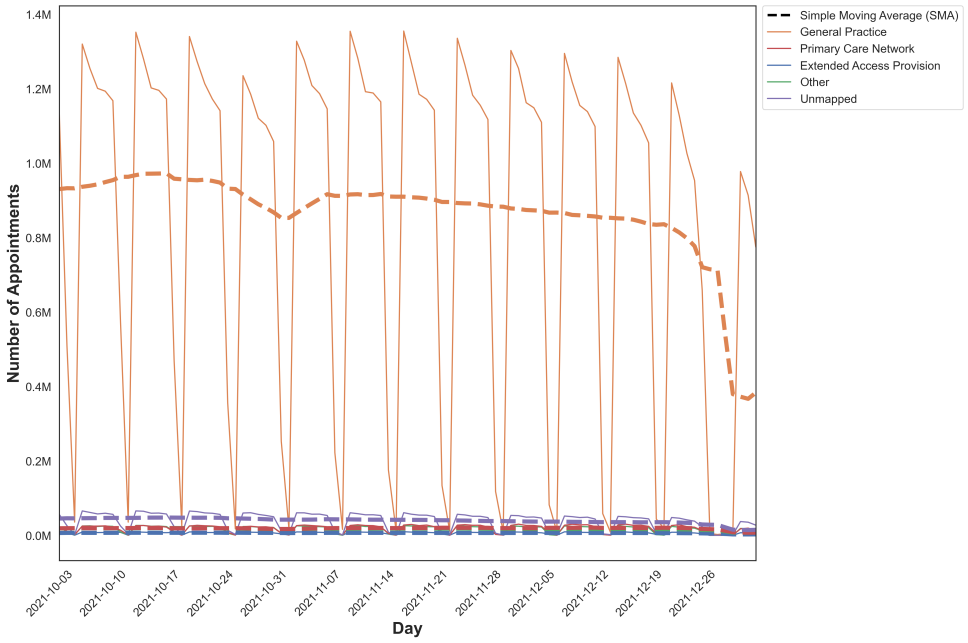

Figure A7: Time series of number of appointments per day throughout autumn 2021.

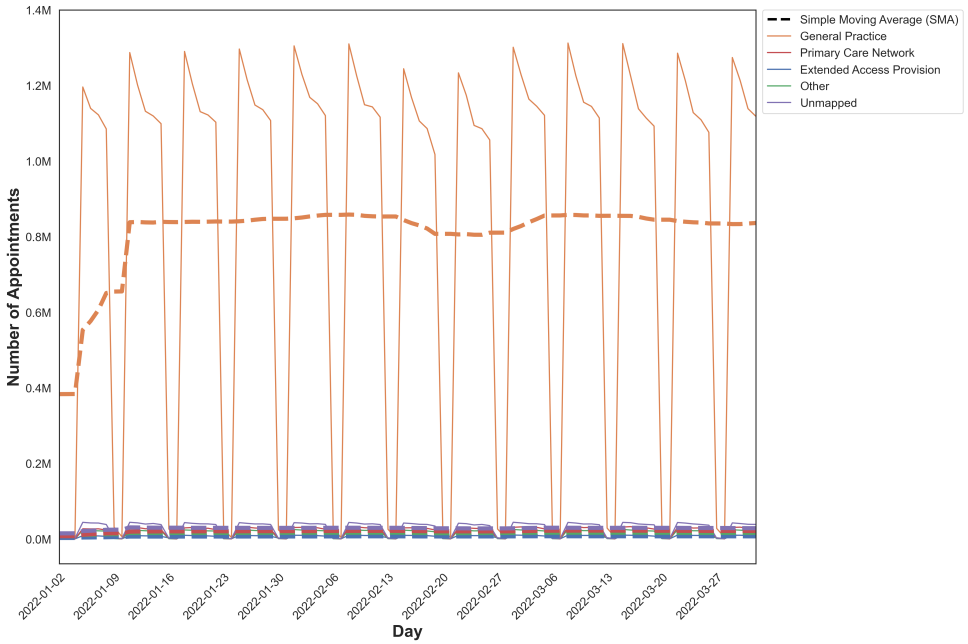

Figure A8: Time series of number of appointments per day throughout winter 2022.

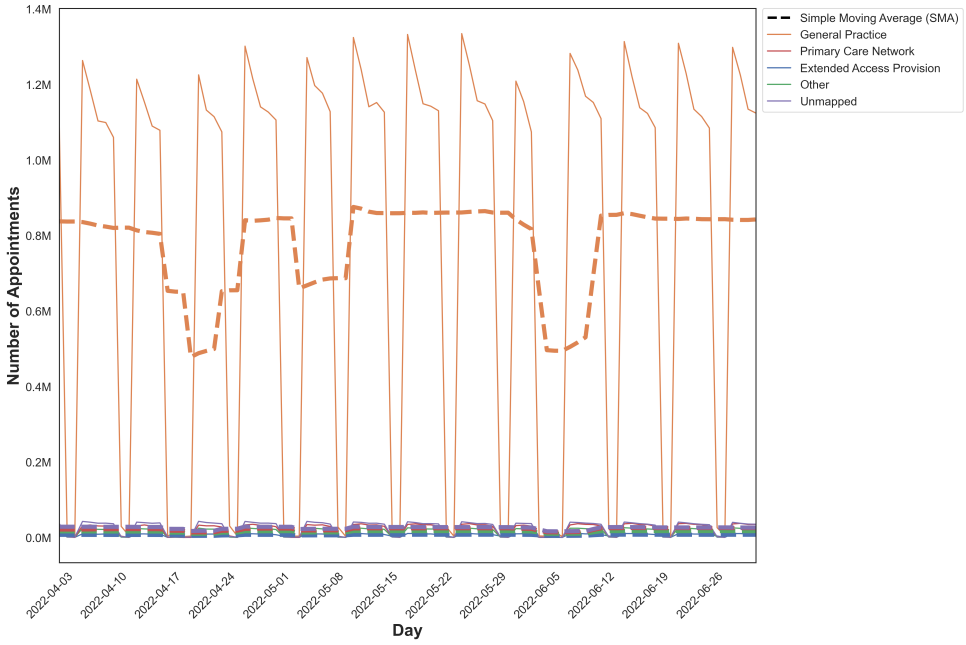

Figure A9: Time series of number of appointments per day throughout spring 2022.

5.2: Analysing the tweet data

Functions:

In [71]:
```
# Loop through the messages, and create a list of values containing the # symbol.
def find_hashtag(tweet_list, separator):
    tags = []
    for y in [x.split(separator) for x in tweet_list.values]:
        for z in y:
            if '#' in z:
                # Change to lowercase.
                tags.append(z.lower())
    return tags
```

Figure A10: Creating a user-defined function that returns words with a **#hashtag** from raw text.

In [72]:
```
# Find hashtags that contain other hashtags to group redundant tags.
def similar_words(word_list):
    # Initialise empty list to fill with values.
    result = [[] for x in range(len(word_list))]

    for word0 in word_list:
        count = -1
        for word in word_list:
            count += 1
            if word0 != word:
                if word0 in word:
                    # Fill empty list with the word(s) contained within.
                    result[count].append(word0)
    return result
```

Figure A11: Creating a user-defined function that returns words that contain another word in a given list.

In [73]:
```
# Function that removes all punctuation from a list of words.
def remove_punctuation(word_list, **kwargs):
    # Optional argument that preserves certain punctuation
    exception = kwargs.get('exception', None)

    # Punctuation exceptions that should NOT be removed.
    if exception is not None:
        for punct in exception:
            my_punctuation = string.punctuation.replace(punct, '')
    else:
        my_punctuation = string.punctuation

    count = -1
    for word in word_list:
        count += 1
        word_list[count] = word.translate(str.maketrans('', '', my_punctuation))

    # Return the output.
    return word_list
```

Figure A12: Creating a user-defined function that removes punctuation from words in raw text.

In [74]:
```
# Function that removes outliers using the 1.5 IQR rule.
def remove_outliers(num_list):
    # Find interquartile range (IQR).
    q3, q1 = np.percentile(num_list, [75 ,25])
    iqr = q3 - q1

    # Find limits to filter out exceedingly small or large outliers.
    lim_condition = ((num_list > (q1 - 1.5*iqr)) & (num_list < (q3 + 1.5*iqr)))

    # Apply limit condition.
    try:
        num_list = num_list[lim_condition]
    except:
        # Attempt to catch incorrect data types.
        num_list = np.array(num_list)
        num_list = num_list[lim_condition]

    # Return output.
    return num_list
```

Figure A13: Creating a user-defined function that removes outliers according to the *1.5 IQR condition*.

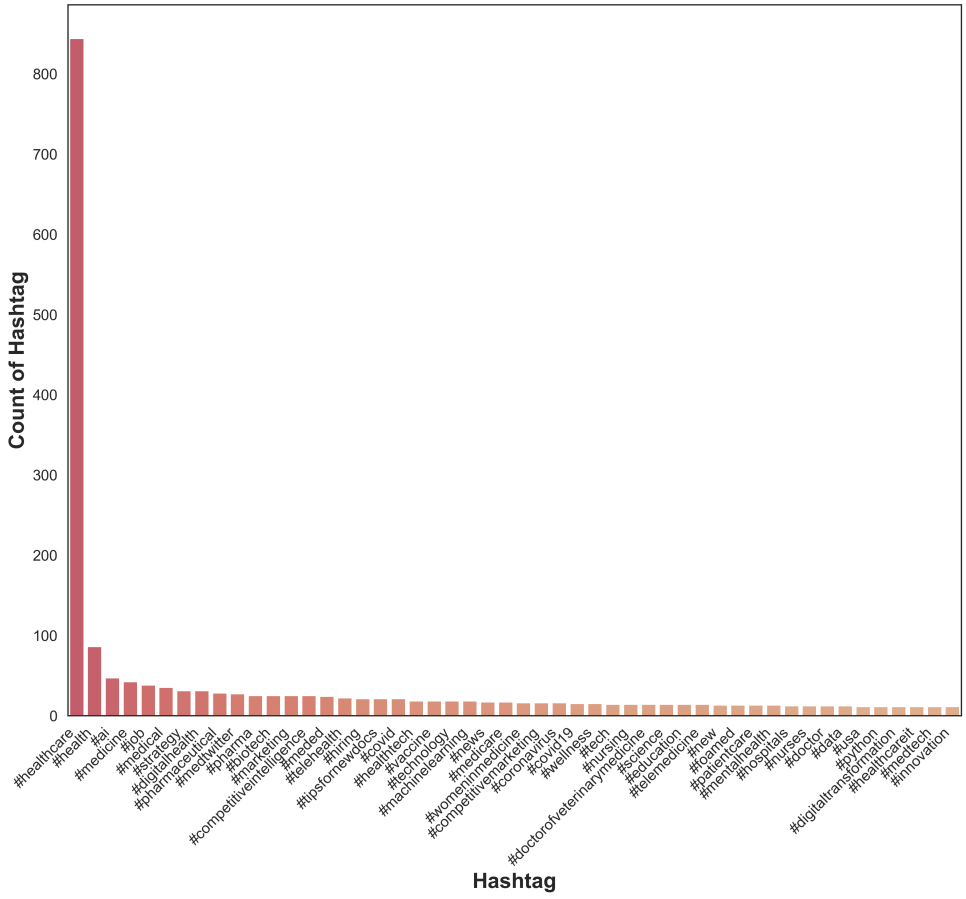

Figure A14: Counting the number of times each **#hashtag** appears in selected tweets.

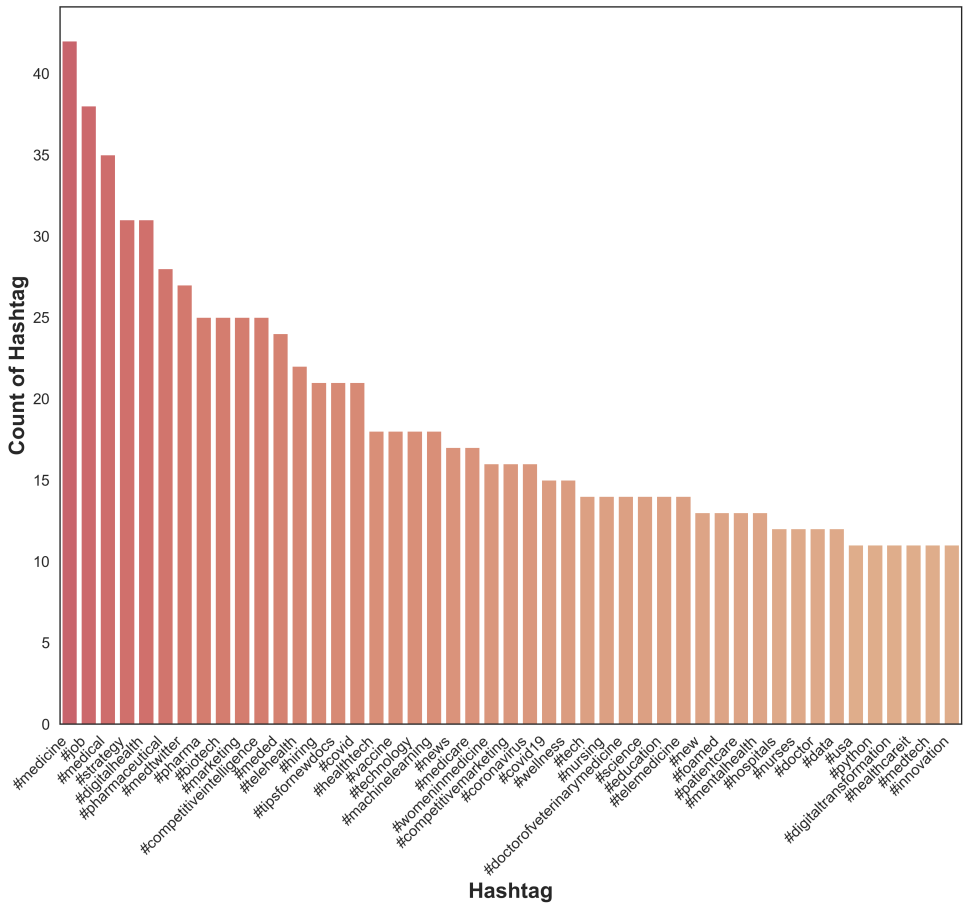

Figure A15: Counting the number of times each **#hashtag** appears in selected tweets, excluding outliers.

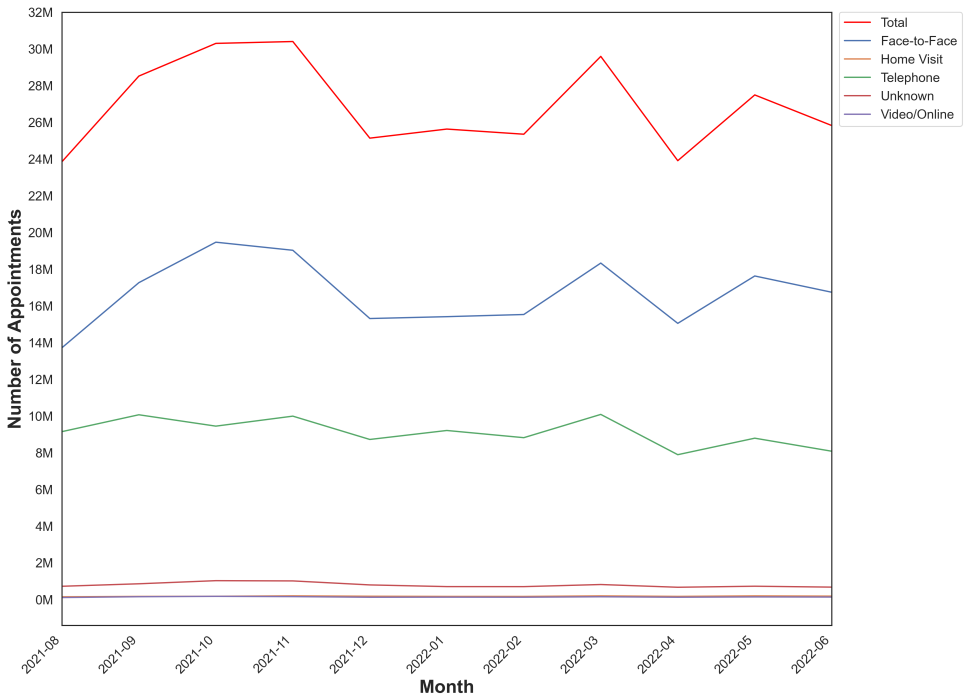

Figure A16: A time series of number of appointments per month, grouped by appointment mode.

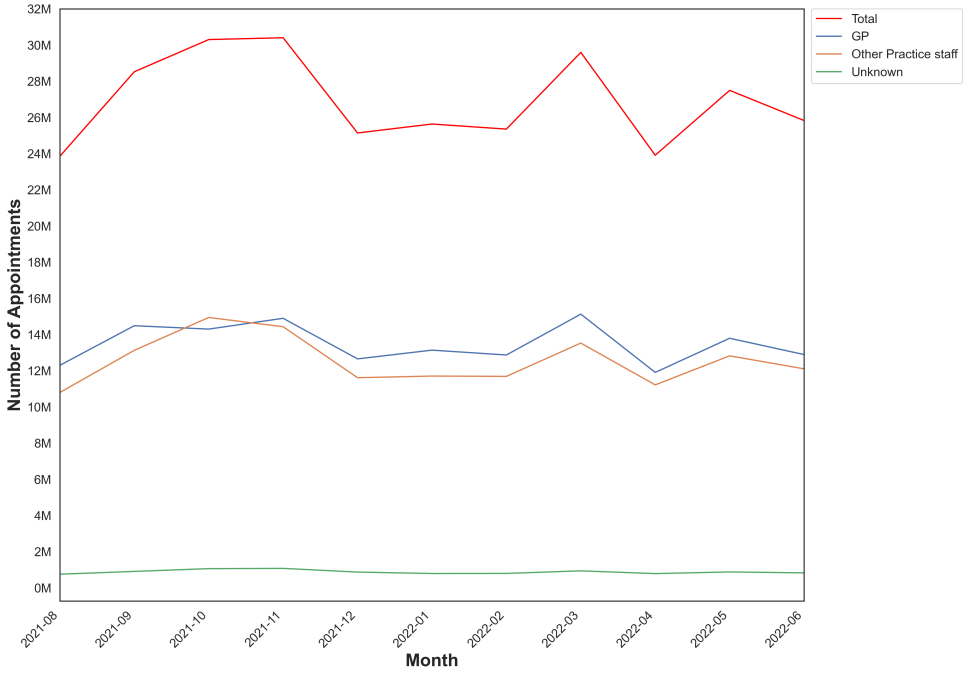

Figure A17: A time series of number of appointments per month, grouped by healthcare provider type.