2025

ACLO Analysis Report

Alyanna (Ming Ming)
Nadine

Table of Contents

Introduction 2
Project Context 2
Project Goal 2
Evaluation Criteria for Data Analysis 2
Data Understanding 3
Data Source & Columns Description 3
Handling Missing, Duplicated and Incorrect Records 4
cardholders dataset 4
checkin_time dataset 5
Descriptive Statistics 8
cardholders dataset 8
checkin_time dataset 8
Additional Data Source 14
Data Analysis Pipeline 20
Data Preparation 20
Create Dataset 20
Create Dummy Variable 22
Encoding 23
Feature Selection 23
Relation with target variable 23
Correlation Matrix 24
Feature Selection 26
Splitting Train Test Set 28
Normalization 28
Splitting Dataset 29
Time Split 29
Random Split 29
Algorithm Candidates 30
Logistic Regression 31
Linear Regression 31
Random Forest 33
XG Boost 34
SVR 37
Poisson Regression 40
Decision Tree 42
Best Performed Algorithm 45

Introduction

Project Context

ACLO, the primary student sports organization in Groningen, currently serves over 19,000 cardholders. While a standard ACLO card grants access to a diverse range of activities including group lessons, courses, and open hours, using the Sports Centre's fitness facilities requires additional purchase of a Fitness Card. Recognizing the low adoption of the Fitness Card, the ACLO seeks deeper insight into current fitness usage.

Project Goal

The primary goal of this project is to develop a predictive model for fitness visitor numbers at the ACLO Sports Centre.
By leveraging historical fitness usage data spanning from 2017 to 2021, this analysis aims to quantify the influence of various internal and external factors on fitness attendance. Specifically, the project will investigate the impact of season, holiday periods, the day of the week, gender, educational institution (University of Groningen vs. Hanze University of Applied Sciences), and exam periods on fitness usage. The project also explores the impact of external events such as COVID-19 lockdowns and weather conditions. The ultimate objective is to provide ACLO with actionable insights that can inform their marketing strategies for the Fitness Card and contribute to informed decisions regarding future service offerings.

Evaluation Criteria for Data Analysis

To assess the performance of the predictive model developed in this project, we will adopt the following evaluation metrics:

1. Mean Absolute Error (MAE)
   This metric will measure the average magnitude of the errors in our predictions. Being less sensitive to outliers, MAE will offer a robust measure of typical prediction errors.
2. Mean Squared Error (MSE)
   This metric will calculate the average of the squared differences between predicted and actual values. It penalizes larger errors more significantly.
3. R-squared
   R-squared will quantify the proportion of the variance in the actual fitness visitor numbers that can be explained by the features included in our model.
4. Adjusted R-squared
   This metric is a modified version of R-squared that adjusts for the number of predictors in the model and the sample size. It helps to prevent overfitting, especially when comparing models with different numbers of predictors.

By employing these evaluation metrics, we aim to comprehensively assess the predictive capabilities of our model and ensure its relevance and reliability for ACLO’s decision-making processes.

Data Understanding

Data Source & Columns Description

The dataset used for this project is provided by ACLO. 4 csv files recording the cardholders data and the other 4 are the check-in times. To make the dataset easier to read, we translated the column names into English.

This is the dataset for cardholders. There's a total of 32,830 records. Description for each column is stated in the table:

client_id	int	The ID number of the client
male	string	Whether the client is male or not, if not then null.
female	string	Whether the client is female or not, if not then null.
card_type	string	The type of card the member has. There are 4 types: 5. ACLO Card 2020-2021 6. ACLO Card 2021-2022 7. Fitness Card 2020-2021 8. Fitness Card 2021-2022

This is the dataset for check-in times. There's a total of 380,785 records. Description for each column is stated below:

client_id	int	The ID number of the client
institution	string	The identity of the client. There are 3 types: 1. Hanze (student) 2. RUG (student) 3. Medewerker Hanze/RUG (employee of the schools)
gender	string	The gender of the client. M for male and V for female.
card_type	string	The type of card the member uses for the fitness service. Filtering the data using the keywords "ACLO card" and "Fitness Card", there are a total of 7 types of cards: 1. ACLO Card 2021-2022 2. Fitness Card 2016-2017 3. Fitness Card 2017-2018 4. Fitness Card 2018-2019 5. Fitness Card 2019-2020 6. Fitness Card 2020-2021 7. Fitness Card 2021/2022
checkin_time	string	The time when the client used the fitness service

As shown below, both the cardholders and check-in times datasets exhibit significant missing values, handling them in each column is a necessary first step later on before analyzing.

```
Index: 32830 entries, 0 to 1370
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   client_id  32830 non-null  int64
 1   male       14903 non-null  object
 2   female     17119 non-null  object
 3   card_type  32830 non-null  object
dtypes: int64(1), object(3)
```

```
RangeIndex: 380785 entries, 0 to 380784
Data columns (total 6 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   Klant nr       380785 non-null  int64
 1   Lid id         102827 non-null  object
 2   Geslacht       380785 non-null  object
 3   Abo bezoek     373011 non-null  object
 4   Incheckdatum   380785 non-null  object
 5   Lid id         253005 non-null  object
dtypes: int64(1), object(5)
```

Handling Missing, Duplicated and Incorrect Records

cardholders dataset

	client_id	male	female	card_type
0	206602	NaN	Vrouw	ACLO Card 2020-2021
1	220965	NaN	Vrouw	ACLO Card 2020-2021
2	221004	Man	NaN	ACLO Card 2020-2021
3	200011	Man	NaN	ACLO Card 2020-2021
4	187611	Man	NaN	ACLO Card 2020-2021

In this dataset, we can see that the genders have separate columns. It is generally not considered a best practice for several reasons, like more inefficient, higher chance to receive errors if the columns are not kept consistent, and more difficult to analyze. We decided to merge the two columns to a new one called “gender”. The new dataset for cardholders is as follows:

	client_id	gender	card_type
0	206602	Vrouw	ACLO Card 2020-2021
1	220965	Vrouw	ACLO Card 2020-2021
2	221004	Man	ACLO Card 2020-2021
3	200011	Man	ACLO Card 2020-2021
4	187611	Man	ACLO Card 2020-2021

```
gender
Vrouw    17119
Man      14903
NaN        808
Name: count, dtype: int64
```

Since “client_id” and “card_type” don’t have missing values, there’s no need for cleaning them. We dropped the missing gender records, returning a dataset with 32,022 records.

```
gender
Vrouw    17119
Man      14903
Name: count, dtype: int64
```

checkin_time dataset

	Klant nr	Lid id	Geslacht	Abo bezoek	Incheckdatum	Lid id
0	168236	Medewerker RUG	M	Fitness Card 2016-2017	2017-08-01 12:49:10	NaN
1	132339	RUG	M	Fitness Card 2016-2017	2017-08-01 12:53:11	NaN
2	141170	RUG	M	Fitness Card 2016-2017	2017-08-01 12:53:34	NaN
3	141170	RUG	M	Fitness Card 2016-2017	2017-08-01 12:54:13	NaN
4	186817	RUG	M	Fitness Card 2016-2017	2017-08-01 13:00:58	NaN

In this dataset, we can see that 'Lid id' was listed twice. Since they showed the exact same information, we combined them into one and translated them for better understanding.

	client_id	institution	gender	card_type	checkin_time
0	168236	Medewerker RUG	M	Fitness Card 2016-2017	2017-08-01 12:49:10
1	132339	RUG	M	Fitness Card 2016-2017	2017-08-01 12:53:11
2	141170	RUG	M	Fitness Card 2016-2017	2017-08-01 12:53:34
3	141170	RUG	M	Fitness Card 2016-2017	2017-08-01 12:54:13
4	186817	RUG	M	Fitness Card 2016-2017	2017-08-01 13:00:58

Based on the original data set, we also discovered that there are 26,632 overlapping records from two of the csv files. We therefore dropped the duplicated records.

```
print(checkinDF.duplicated().sum())
```
✓ 0.1s
```
26632
```

```
checkinDF = checkinDF.drop_duplicates()
print(checkinDF.duplicated().sum())
```
✓ 0.2s
```
0
```

The information on the dataset is shown below. Both “institution” and “card_type” have missing values. We decided to further look into them.

```
RangeIndex: 354153 entries, 0 to 354152
Data columns (total 6 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   index         354153 non-null  int64
 1   client_id     354153 non-null  int64
 2   institution   330984 non-null  object
 3   gender        354153 non-null  object
 4   card_type     346442 non-null  object
 5   checkin_time  354153 non-null  object
dtypes: int64(2), object(4)
```

The following shows part of the unique values of “institution”. It is very messy.

```
print(checkinDF["institution"].unique())
```
✓ 0.0s
```
['Medewerker RUG      ' 'RUG       ' nan 'Hanze       ' 'Hanze      '
 '         ' ' ' 'GORM' 'CRUGET' '     ' 'WIWI' 'GEEJ' '       ' 'ETGE'
 'DROK' 'OLTI' 'Medewerker RUGORUGHanze' '   ' 'WITF' '  ' '      ' 'JEVA'
 'VALE' 'UIWI' 'VWIL' 'Hanze ' '    ' 'DUJMedewerker RUG' 'KAJA'
 'RUGCHanzeHanze' 'MEAD' '        ' 'RORA' 'Medewerker RUGARN' 'JEEU'
 'Medewerker RUG/Hanze        ' 'Medewerker RUG/Hanze          '
 'Medewerker RUG/Hanze           ' 'RUG           '
 'Medewerker RUG/Hanze       ' 'RUG             ' 'Hanze            '
 'RUG          ' 'RUG            ' 'Medewerker RUG/Hanze         '
```

There are two things to do: remove redundant spaces, drop null values, and drop records with special characters. Also, after confirming with the instructor, values like MOOM, AJAA are actually the workers’

id from Hanze and RUG. Hence, we decided to create a mask and label them "Medewerkers RUG/Hanze". The following photos demonstrate the progress.

```
#clean data
checkinDF["institution"] = checkinDF["institution"].str.strip()
checkinDF.dropna(subset=["institution"], inplace = True)
print(checkinDF["institution"].unique())
```

✓ 0.1s

```
'Medewerker RUG' 'RUG' 'Hanze' '' 'GORM' 'CRUGET' 'WIWI' 'GEEJ' 'ETGE'
'DROK' 'OLTI' 'Medewerker RUGORUGHanze' 'WITF' 'JEVA' 'VALE' 'UIWI'
'VWIL' 'DUJMedewerker RUG' 'KAJA' 'RUGCHanzeHanze' 'MEAD' 'RORA'
'Medewerker RUGARN' 'JEEU' 'Medewerker RUG/Hanze'
'MedewerkerRUG RUG/HanzeanzeARN' 'DUJMedewerkerRUG RUG/Hanzeanze' 'BNAN'
'KILI' 'JENRUG' 'BONI' 'HanzeOHanzeO' 'VEHanzeW' 'VRJO' 'RUGAL_dili'
'MORN' 'A.KoRUGter@rug.nl' 'TARI' 'Medewerkers RUG/Hanze'
```

```
#remove unnecessary and find out all workers
wrongMask = ~checkinDF["institution"].str.contains("_|-|@", na=False)
workerMask = (~checkinDF["institution"].isin(["RUG", "Hanze"])) & (
    (checkinDF["institution"].str.len() == 4) | (checkinDF["institution"].str.contains("Medewerker|Hanze|RUG"))
)

checkinDF = checkinDF[wrongMask]

workerDF = checkinDF[workerMask]

instituteCount = workerDF["institution"].value_counts()
print(instituteCount.to_string())
```

✓ 0.2s

```
#make all workers into category Medewerker
checkinDF.loc[workerMask, "institution"] = "Medewerker RUG/Hanze"
print(checkinDF["institution"].unique())
print(checkinDF["institution"].value_counts(dropna=False))
```

✓ 0.0s

```
['Medewerker RUG/Hanze' 'RUG' 'Hanze' '']
institution
RUG                   209971
Hanze                  78432
Medewerker RUG/Hanze   33908
                        6342
Name: count, dtype: int64
```

Noticing there are still records with empty values counted, we applied replace and drop those records again, returning a dataset with 322,311 records.

```
#remove the annoying ''
checkinDF["institution"].replace('', np.nan, inplace = True)
checkinDF.dropna(subset=["institution"], inplace = True)
print(checkinDF["institution"].value_counts(dropna=False))
```

✓ 0.0s

```
institution
RUG                   209971
Hanze                  78432
Medewerker RUG/Hanze   33908
Name: count, dtype: int64
```

Next, is to handle missing values in “card_type”. We listed out the unique values first.

```
print(checkinDF["card_type"].unique())
```
✓ 0.0s

```
['Fitness Card 2016-2017' 'Fitness Midseason Card 2016-2017'
 'Fitness Summer Card 2016-2017' 'Fitness Free Card 2016-2017'
 'Power Card NorthsideBB 2016-2017' 'Power Card 2016-2017'
 'Fitness Card 2017-2018' 'Fitness topsport 2016-2017'
 'Fitness Free Card 2017-2018' 'Fitness ExchangeStud 2017-2018'
 'Fitness topsport 2017-2018' 'Power Card 2017-2018'
```

Since this project will only focus on ACLO Card and Fitness Card, we decide to include only those records after dropping the empty values, returning a dataset with 238,462 records.

```
checkinDF.dropna(subset=["card_type"], inplace=True)
```
✓ 0.0s

```
#fitness usage by each card type
checkinDF = checkinDF[checkinDF["card_type"].str.contains("ACLO Card|Fitness Card")]
print(checkinDF["card_type"].unique())
```
✓ 0.1s

```
['Fitness Card 2016-2017' 'Fitness Card 2017-2018'
 'Fitness Card 2018-2019' 'Fitness Card 2019-2020'
 'Fitness Card 2020-2021' 'Fitness Card 2021/2022' 'ACLO Card 2021-2022']
```

```
RangeIndex: 238462 entries, 0 to 238461
Data columns (total 6 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   index         238462 non-null  int64
 1   client_id     238462 non-null  int64
 2   institution   238462 non-null  object
 3   gender        238462 non-null  object
 4   card_type     238462 non-null  object
 5   checkin_time  238462 non-null  object
dtypes: int64(2), object(4)
```

During the cleaning process, we discovered that there are also incorrect values in “gender”. We removed them, returning 231,214 records for the checkin_time dataset.

```
#remove fout(error) data
print(checkinDF["gender"].value_counts(dropna=False))
foutMask = checkinDF["gender"] == "fout"
checkinDF = checkinDF[~foutMask].reset_index(drop=True)

print(f"After cleaning {checkinDF['gender'].value_counts(dropna=False)}")
```
✓ 0.0s

```
gender
M       189170
V        42044
fout      7248
Name: count, dtype: int64
After cleaning gender
M    189170
V     42044
Name: count, dtype: int64
```

Lastly, we modified the data type of “checkin_time”. We changed it from string to datetime for better data handling.

```
print(type(checkinDF.loc[0, "checkin_time"]))
```
✓ 0.0s

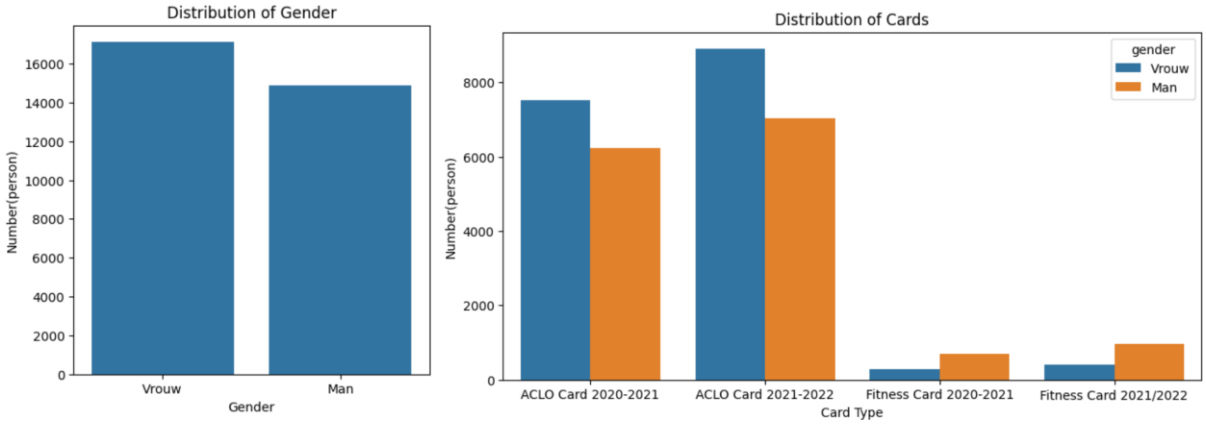
```
<class 'str'>
```

```
#make string time into datetime
checkinDF["checkin_time"] = pd.to_datetime(checkinDF["checkin_time"])
```

Descriptive Statistics

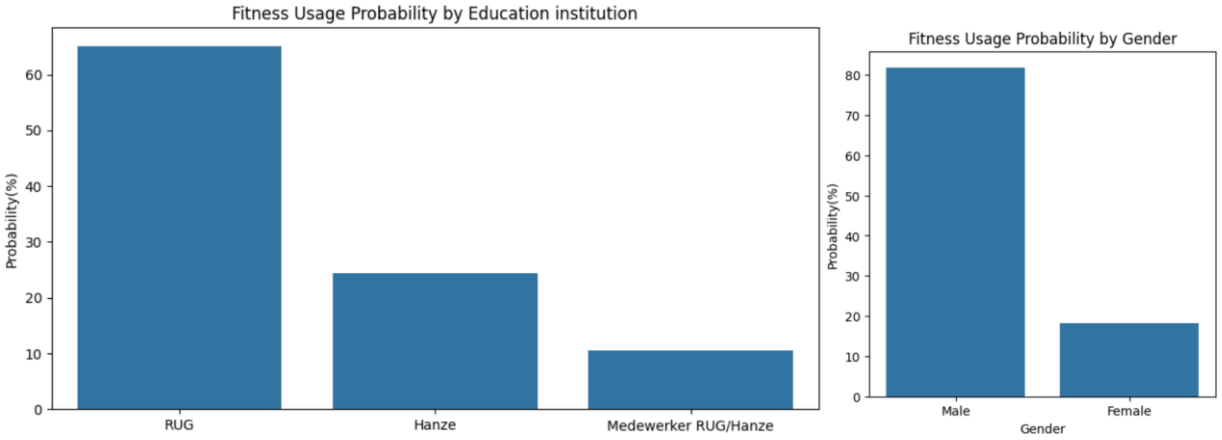
cardholders dataset

The two charts below illustrate the distribution of gender of the cardholders. Overall, the cardholder base is nearly evenly split by gender, with a slight female majority. However, if we take a closer look by the card types as the right chart indicates, the ACLO card has higher female cardholders, while the Fitness Card is mostly male. And consistent with the Introduction, the number of Fitness Card holders is considerably lower than that of ACLO Card holders.

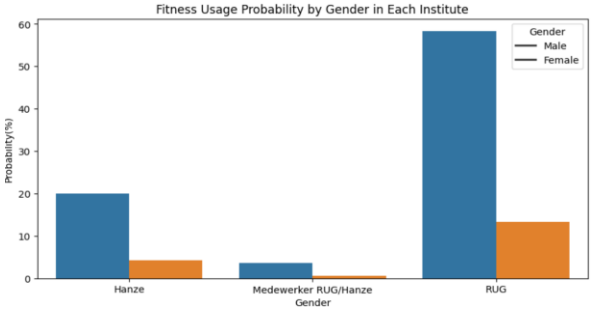


checkin_time dataset

For the checkin_time dataset, we started by analyzing “institution” and “gender”. To better compare the differences, we change the units into percentages. The data indicates that students from RUG represent the highest percentage of fitness users, followed by students from Hanze and then Medewerkers. Previously, we mentioned there is a higher portion of male in the fitness card. In the right chart, we can also see that the fitness usage is mostly male, around 80% of the total users.



When we look at both school/work and gender together, men still use the fitness area more at all institutions. What's interesting is that the difference between how many men and women use the fitness area is smallest for Medewerker and biggest for RUG students.

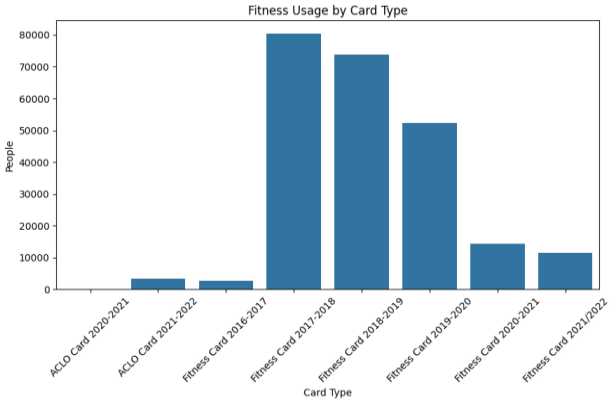


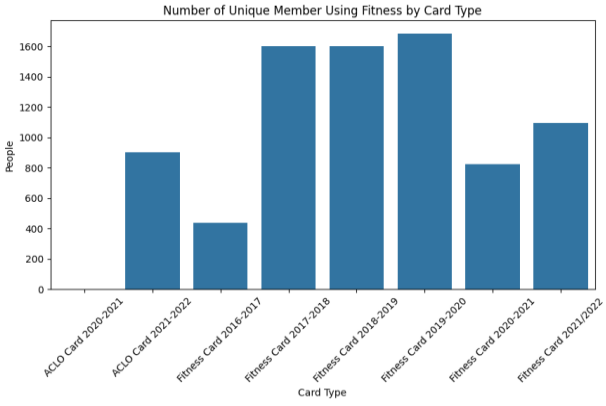
institution	gender	frequency	percentage
Hanze	M	46128	19.950349
Hanze	V	9728	4.207358
Medewerker RUG/Hanze	M	8202	3.547363
Medewerker RUG/Hanze	V	1458	0.630585
RUG	M	134840	58.318268
RUG	V	30858	13.346078

From the above findings, we can tell that even though RUG students use the fitness area the most overall, it's mostly the male students. And also, the Medewerker has the most balanced usage between men and women among the other two. This suggests that promotional strategies and service offerings might benefit from being tailored to address the specific needs and preferences of different user groups and genders within the ACLO community. Understanding these distinct usage patterns will be crucial for the ACLO to effectively promote the Fitness Card and optimize its services for all members.

Next, we wanted to take a closer look into the “card_type” column of the checkin_time dataset. The left chart is the overall fitness usage of different cardholders, while the right one shows the unique number of fitness users from each card type. The chart on the left shows that people with a Fitness Card in 2017-2018 used the fitness area the most, and their usage went down each year after that. It also shows that the lowest usage was for ACLO cardholders in 2020-2021 and 2021-2022 and Fitness Card holders in 2016-2017.

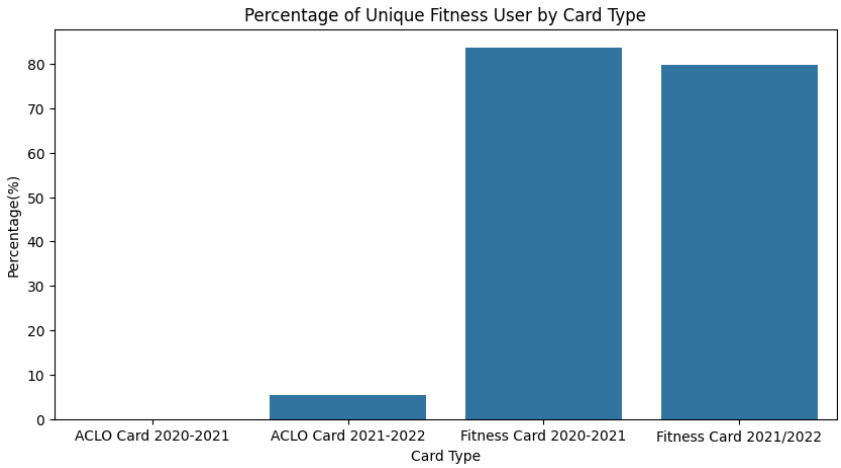
Interestingly, despite Fitness Card 2016-2017 still being the second lowest on the right chart, ACLO 2021-2022 has slightly more unique users than the Fitness Card in 2020-2021, and about 200 fewer users compared to the Fitness Card in the same year (2021-2022)!



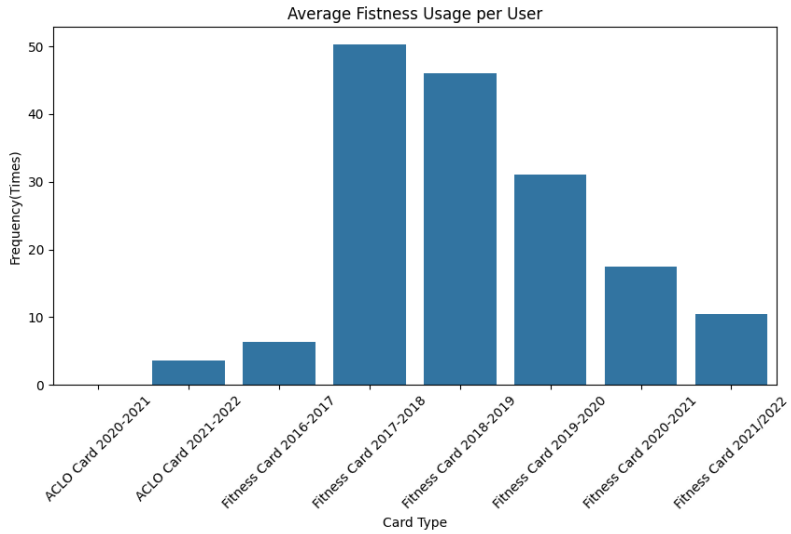


We also wanted to know among all cardholders from the cardholders dataset, how many of them actually use the fitness area. Since the cardholders dataset doesn’t have all of the card types in checkin_time dataset, the left chart only compares the cards between 2020-2021 and 2021-2022. We

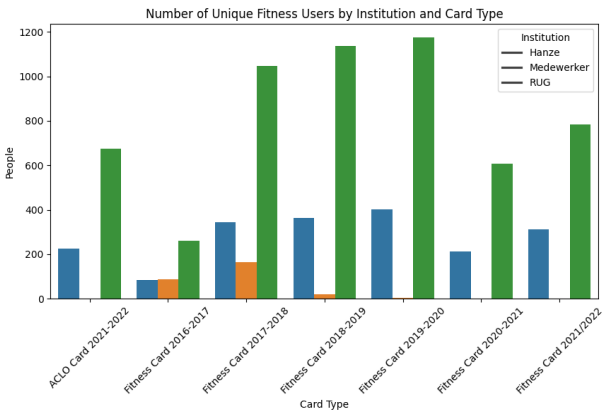
can tell that more than 80% of the Fitness Card 2020-2021 holders use the fitness area, followed by nearly 80% of the Fitness Card 2021-2022 holders. Both ACLO Cards got the lowest percentage of fitness usage, with 0% and less than 10% of them used.

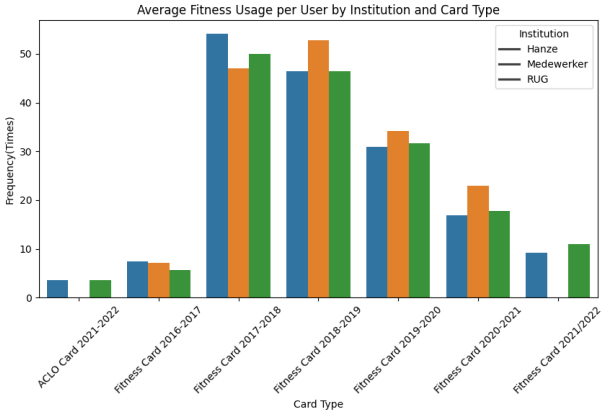


The chart below demonstrates the average fitness usage per user from each card type. We can see that the distribution is fairly similar to the first chart, with Fitness Card 2017-2018 the highest, and decreased each year after that.



The combination of card types and institutions is shown below. We already know that RUG has the highest fitness usage among the other two, and this result can be seen by the chart on the left as well– RUG has the highest fitness usage across all card types. But if we check it by the average usage per person among the card types, the result turns out to be interesting. RUG doesn't always hold the highest average usage. It is the Hanze students and Medewerker, especially the latter, that got higher average usage.

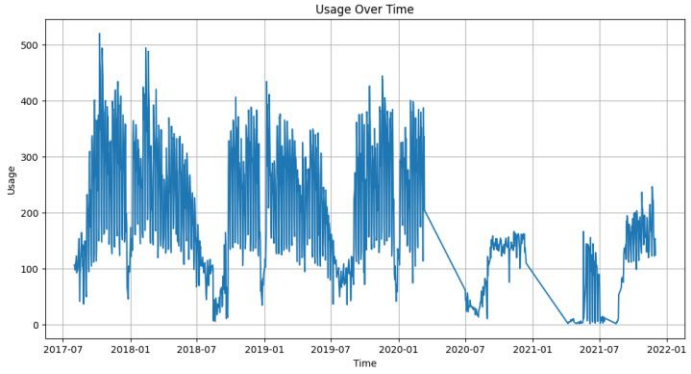




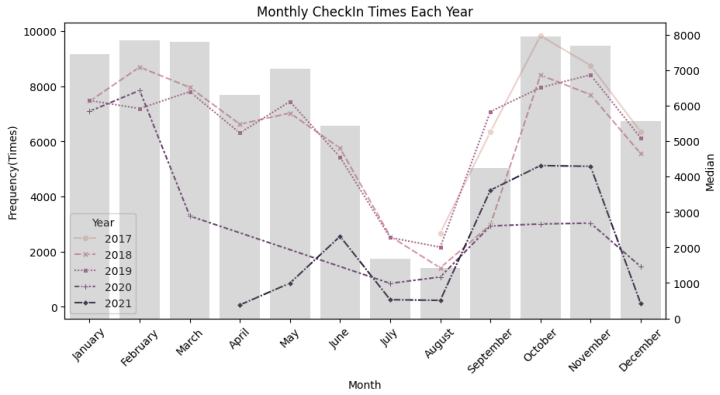
Based on the findings presented above, we can draw the following key insights:

1. Declining Engagement Among Fitness Card Holders:
   Analysis of Fitness Card holders reveals a concerning trend of decreasing overall fitness usage and a reduction in the average usage per individual over time. This suggests a potential need to re-evaluate the value proposition or engagement strategies for this specific membership type.
2. ACLO Cardholder Potential:
   Despite the relatively low overall fitness usage among standard ACLO cardholders, with less than 10% of them using the fitness area in 2021-2022, the number of unique ACLO cardholders accessing the fitness facilities is not the lowest compared to other card types across different years. This indicates potential within the general ACLO membership base that could be leveraged with appropriate incentives or integrated access options. For instance, as noted earlier, the number of unique ACLO card users in 2021-2022 slightly surpassed that of Fitness Card holders in 2020-2021 and was only marginally lower than Fitness Card holders in the same year (2021-2022), highlighting a considerable pool of individuals with some level of interest in fitness.
3. Higher Average Fitness Usage are Hanze and Medewerker:
   When we look at how often individual card holders visit the fitness area on average, the results are interesting. RUG students, despite being the largest overall user group, don't consistently have the highest average usage per person. Instead, Hanze students and, notably, Medewerkers tend to use the fitness facilities more frequently on average. This suggests that while RUG students might represent the biggest user base, those from Hanze and the ACLO staff who choose to use the fitness area are more regular and intensive users.

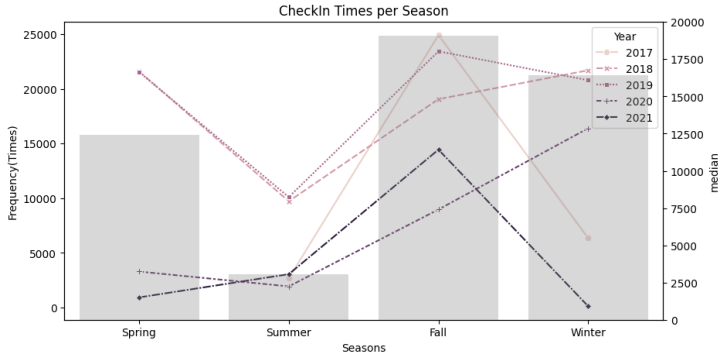
After discovering the interesting pattern of the card types, we will further move on to the “checkin_time” column. This line plot visualizes the total fitness usage over time. From it, we can observe two distinct periods where usage data is absent and that the line is very noisy. To better extract information from it, we decided to look into the date by month, season, and day of the week.



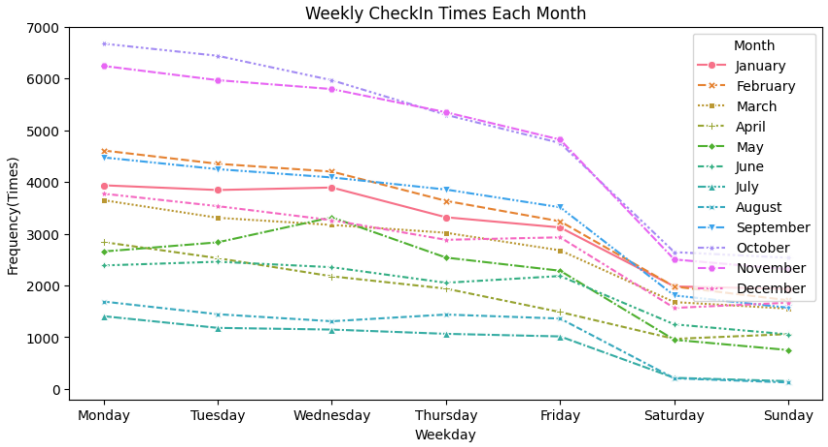
This plot shows the monthly fitness usage every year. The bar plot is the median of the total usage of that month and each line represents the fitness usage of a different year. Surprisingly, the fitness usage between July and August is the lowest, a little bit of an “M” shape.



The result by season is more distinctive. We can really tell that summer got the lowest fitness usage among the year, while autumn got the highest. Since most cardholders are students, we suspect that maybe it has some relation with the fact that fall is usually the start of the semester year, and that summer is the end of it.



Fitness usage throughout the week also follows a clear pattern, peaking on Monday and steadily decreasing in the following days.



These three plots tell a very interesting story. We can conclude several key patterns and potential drivers of fitness usage at the ACLO:

1. Summer Decrease:
   The consistent low usage during July and August strongly suggests a significant seasonal effect, likely due to summer vacation, reduced student presence in Groningen, and potentially warmer weather encouraging outdoor activities.
2. The "Fresh Start" Effect:
   The strong correlation between the highest usage occurring on Monday (the start of the week), in the fall (the start of the academic year), and around February and October (the start of the semester) points to a "fresh start" or "new beginnings" phenomenon influencing fitness engagement. We therefore propose that individuals are more motivated to begin or resume fitness routines at the start of new cycles.

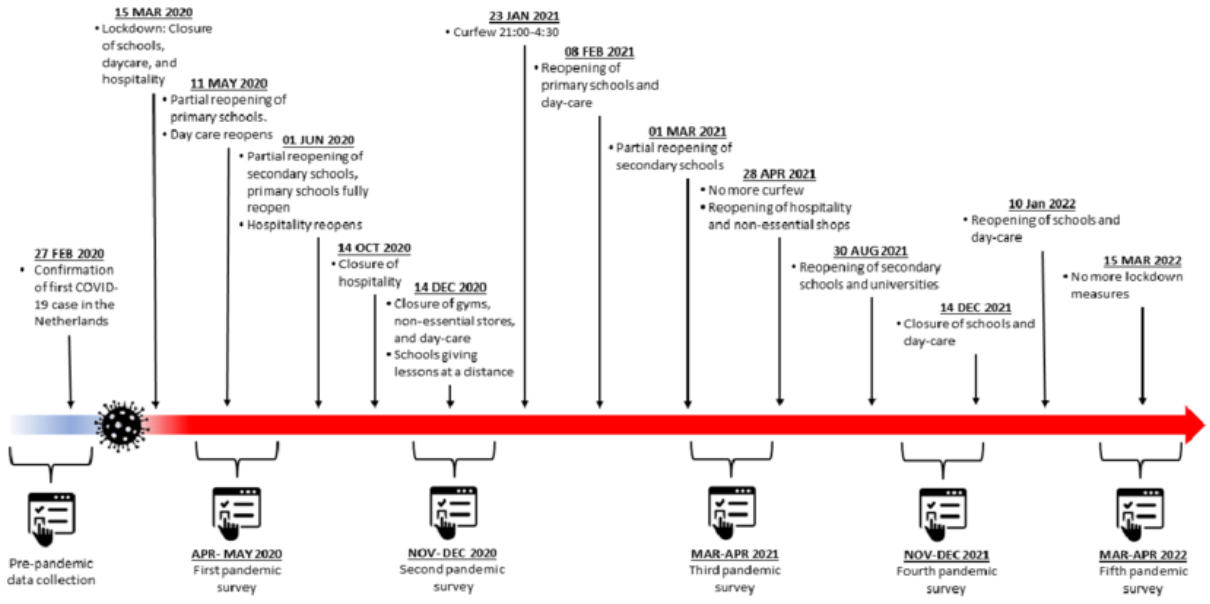
Furthermore, these insights can inform some strategic considerations for the ACLO, for instance, focusing on "Fresh Start" moments. Marketing efforts could be strategically timed with the beginning of the week (Mondays) and the start of each semester (especially Fall) to leverage this inherent motivation. In essence, the above findings provide a valuable foundation for a deeper understanding of user behavior and can guide ACLO in developing more effective strategies to promote fitness engagement throughout the year.

Additional Data Source

To gain a deeper understanding of the trends in fitness usage, we cross-referenced the fitness usage data with information on holidays, exam periods, COVID-19 lockdowns, and weather conditions to identify potential explanations for the observed patterns.

The holiday and exam periods were based on RUG's academic calendar. As a student here in Groningen, we noticed that the exam and holiday periods are slightly different between the two schools. However, the accessible academic calendar of Hanze is only the ones from 2023. Since the differences are not that huge, to maintain consistency, we decided to adopt the academic calendar of RUG. Resources are available through their school website. Based on the data, we created a csv file with date being the first column, holiday as the second, and exam as the third.

During the Covid-19 lockdown period, we believe the fitness usage was also affected. To address a more comprehensive aspect of the prediction, we decided to integrate the data as well. The lockdown period in the Netherlands is based on the below graph by *Zijlmans, J., Tieskens, J. M., van Oers, H. A., Alrouh, H., Luijten, M. A., de Groot, R., et al. (2023)*. The lockdown period is therefore recorded as the fourth column of the csv file.



While analyzing the fitness usage by month, we assumed that one of the reasons for the discretion during July and August is due to the temperature. Hence, we decided to check by looking into the temperature data of each day. Thanks to the help of one of our Dutch classmates, we found a weather website (https://weerstatistieken.nl/) recording the temperature of the Netherlands from 2017. Since they don't specifically have the data of Groningen, we used the temperature data from De Bilt, the nearest city in their records.

The four features are stored in two csv files, calendar.csv and weather.csv. The column explications are as follow:

calendar.csv

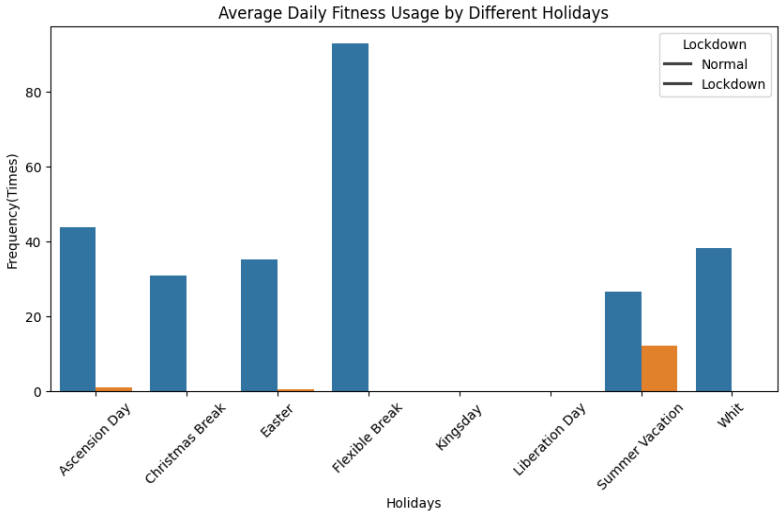
date	string	Date from 01-01-2017 to 31-07-2025
holiday	string	The holiday name of the day belongs to, null if the day is not a holiday. There are 8 holidays in total: 1. Ascension Day (40 days after Easter) 2. Christmas Break (Mid-December to early January) 3. Easter (Late March or April, varies yearly) 4. Flexible Break (RUG's flexible adjustment in accordance with national holidays) 5. Kingsday (27th of April) 6. Liberation Day (5th of May) 7. Summer Vacation (Mid July to end of August) 8. Whit (50 days after Easter)
Exam	float	Whether the day is during an exam period. 1 for yes and 0 for no.
Lockdown	float	Whether the day is during the Covid-19 lockdown period. 1 for yes and 0 for no.

weather.csv

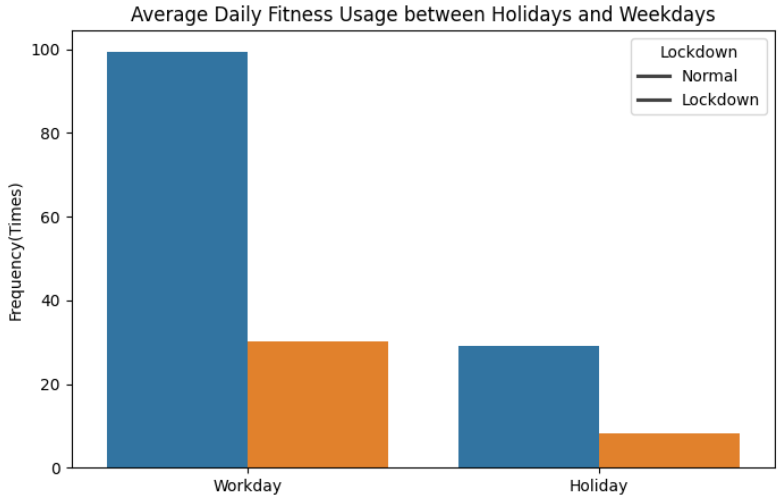
date	string	Date from 01-01-2017 to 31-07-2025
mean_temp	float	The mean temperature of the day, by celsius.
diff_temp	float	The difference between the minimal temperature and the maximal temperature of the day, by celsius.
max_temp	float	The maximal temperature of the day, by celsius.
min_temp	float	The minimal temperature of the day, by celsius.

To improve data handling, we converted the "date" column in both datasets to datetime format. While the "Exam" and "Lockdown" columns are currently numerical, we will later treat them as categorical variables using tokenization, so we left them unchanged for now.
Recognizing that the COVID-19 lockdowns likely impacted fitness usage patterns during both holidays and exam periods, we decided to analyze the influence of holidays and exams by differentiating between periods that occurred during and outside of the lockdown.
The graph below is the fitness usage of different holidays. To avoid longer holidays skewing the results with higher total usage, we calculated and compared the average daily usage for each holiday period. From the graph, we can see that there indeed is nearly no one using the fitness area during the COVID-19 lockdowns. In times when there are no lockdowns, Flexible Break has the highest fitness usage. The reason why there is no usage on Kingsday and Liberation Day is probably due to ACLO not opening during those days. Interestingly, despite having longer duration, summer vacation and Christmas vacation are the two lowest on the chart. This finding aligns with one of the assumptions we made

previously, about the possibility of students not being in Groningen during those periods which causes a decrease in fitness usage.



This graph shows the overall fitness usage between holiday periods or not. To avoid more days in workday skewing the results with higher total usage, we calculated and compared the average daily usage for workday and holidays. Whether there is a lockdown seems to not affect the fact that people tend to participate in fitness service when it is a workday. This further supports our idea that the number of students in Groningen affects fitness usage, and holidays are a key factor in whether students stay or leave.

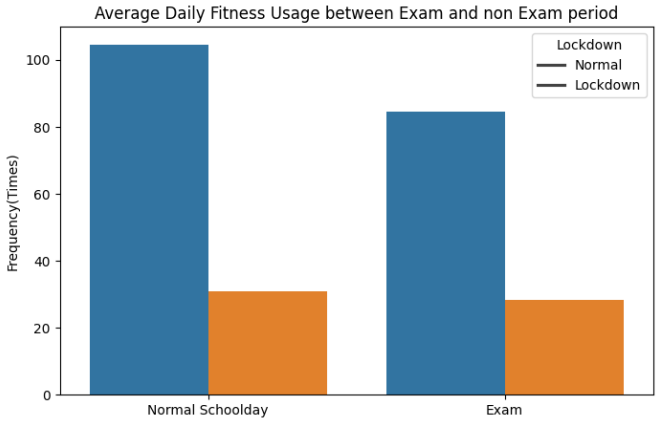


From the above two graphs, we can say that holiday periods, particularly longer breaks like summer and Christmas vacations, strongly suggest a significant decrease in the number of users present in Groningen during these times. On the other hand, shorter breaks like the Flexible Break exhibit the highest average daily usage among other holidays when not impacted by lockdowns, indicating that students and staff present in Groningen during these shorter breaks are more likely to use the fitness facilities. Finally, the persistent higher average daily usage on workdays, regardless of lockdown status, reinforces the idea

that the presence of the regular student and staff population in Groningen is a primary driver of fitness center attendance.
Therefore, regarding ACLO decision-making, we believe these insights suggest the following:

1. Promotional Opportunities:
   Focus marketing efforts and potentially introduce special programs during shorter breaks to capitalize on the higher engagement of those remaining in Groningen.
2. Student Presence:
   Strategies aimed at encouraging students to remain in Groningen during longer breaks could potentially boost fitness usage during those typically low periods.

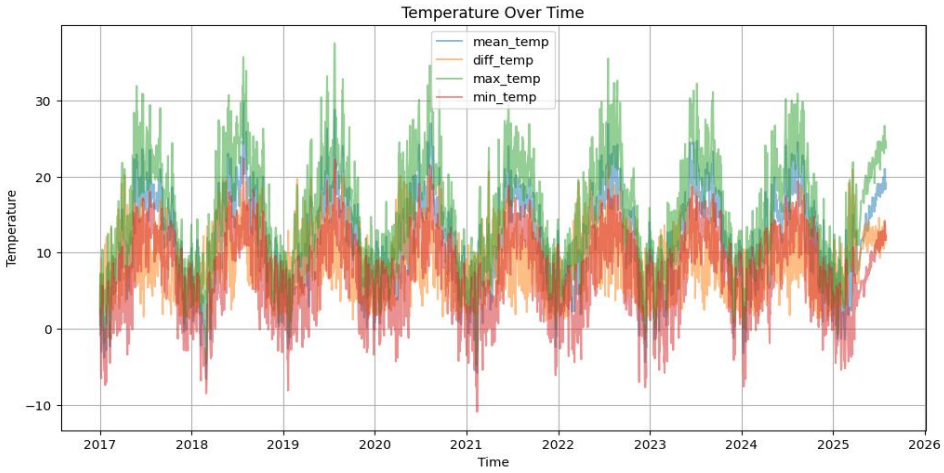
The second one we're examining is the "exam" column from the calendar dataset. As the chart below presents, the fitness usage between whether it is during the exam period is not significantly different, especially during the Covid-19 lockdowns, the usage levels were consistently low regardless of exam schedules.



This finding suggests a few key things:

1. Exams are not Major Barriers:
   Students might adjust their study schedules or stress-relief activities in a way that doesn't reduce their fitness habits.
2. Potential for Consistent Programming:
   ACLO may not need to significantly alter their fitness class schedules or offerings specifically around exam times, as usage appears relatively stable.

The last one we're analyzing is the weather dataset. Here is an overview for the temperature data throughout the years. The temperature follows a fairly predictable trend annually.



Previously, we proposed that one of the reasons for the discretion during Summer is due to the temperature. To address this assumption, we will need to create a dataset with temperature and usage.

```
tempUsage = byWeather.merge(usage, left_on = "date", right_on = "checkin_time")
display(tempUsage.head())
```

✓ 0.0s

	date	mean_temp	diff_temp	max_temp	min_temp	checkin_time	usage
0	2017-08-01	17.8	10.2	22.7	12.5	2017-08-01	106
1	2017-08-02	17.6	8.5	22.0	13.5	2017-08-02	106
2	2017-08-03	19.0	6.0	22.5	16.5	2017-08-03	97
3	2017-08-04	18.3	6.4	21.8	15.4	2017-08-04	101
4	2017-08-07	16.8	13.8	22.6	8.8	2017-08-07	122

In order to better interpret, we decided to transform these numerical temperature data into categorical data, and group by the temperature category.

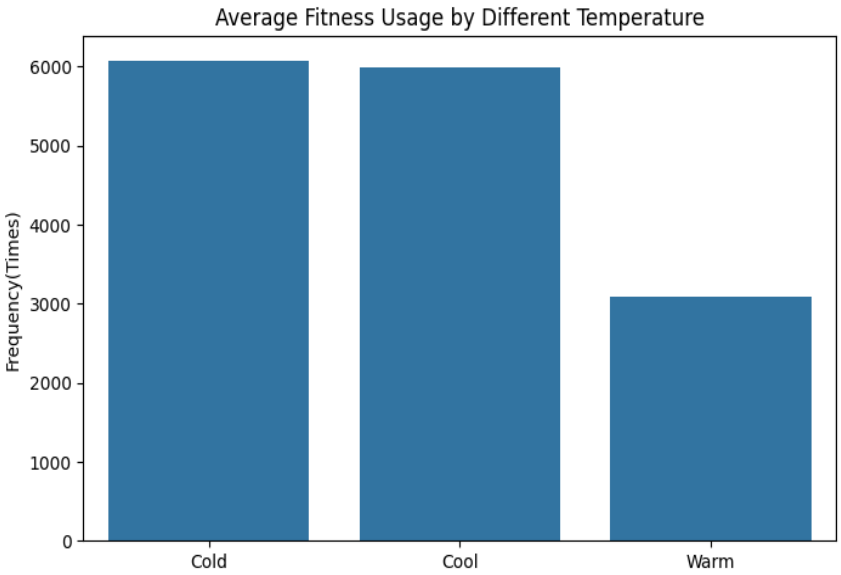
```
def getTemp(temp):
    if temp < 0:
        return "Freeze"
    elif temp < 10:
        return "Cold"
    elif temp < 15:
        return "Cool"
    elif temp < 20:
        return "Warm"
    else:
        return "Hot"
```

```
coldUsage = tempUsage.groupby("mean_temp")["usage"].mean().reset_index(name="usage")
display(coldUsage)
```

✓ 0.0s

	mean_temp	usage
0	Cold	232.304527
1	Cool	204.459547
2	Freeze	233.240000
3	Hot	102.666667
4	Warm	132.354232

As the chart described, we can clearly see that the warmer the weather gets, the lower the fitness usage is. While this doesn't definitively prove that temperature directly affects usage, it strengthens the idea that it might. We suspect people are less likely to do indoor activities and prefer being outside when it's warmer.



For ACLO, the potential influence of temperature on fitness usage suggests a few key considerations:

1. Seasonal Programming Adjustments:
   Be prepared for a likely drop in indoor fitness attendance during warmer months and plan accordingly with staffing and class schedules.
2. Outdoor Offerings:
   Consider expanding or promoting outdoor fitness options or partnerships during warmer periods to fit members' preference for outdoor activities.
3. Marketing and Promotion:
   When marketing the indoor facilities during warmer months, focus on aspects that outdoor activities can't easily replicate.
4. Consider Flexible Membership Options:
   Maybe offering some flexible membership options that align seasonal preferences. For example, a reduced cost "summer outdoor access" membership or temporarily pausing indoor access during peak outdoor season.

We also recommend ACLO continue tracking fitness usage in relation to temperature in Groningen to gain a more precise understanding of the local correlation. This will help refine strategies over time.

In summary, the data-driven insights obtained from cardholders, checkin_time, calendar and weather dataset provide a clear roadmap for the ACLO to enhance fitness engagement. To capitalize on the trends observed, we recommend using different approaches:

1. Personalized Engagement Strategies
   Develop targeted marketing and program offerings that align with the distinct preferences and usage patterns of RUG students (addressing the gender imbalance), Hanze students, and Medewerkers.
2. Re-evaluate the Fitness Card

Re-evaluate the value proposition and implement proactive strategies to improve retention and increase usage among Fitness Card holders.

3. Activating Standard ACLO Members
   Explore and implement accessible pathways or incentives to convert the significant number of unique standard ACLO card users with some fitness interest into regular fitness users.
4. Strategic Calendar Alignment
   Focus on the "fresh start" effect at the beginning of semesters and weeks with timely promotions and programs, while adjusting resources and potentially offering targeted activities during shorter breaks where engagement is higher.
5. Embracing Seasonal Flexibility
   Proactively develop and promote attractive outdoor fitness options during warmer months, while highlighting the unique benefits of the indoor facilities.

By implementing these recommendations, we believe ACLO can optimize its resources, broaden its reach, and foster a more consistently engaged fitness community throughout the year in Groningen.

Data Analysis Pipeline

Data Preparation

Create Dataset

It will be difficult to make time-based predictions from four separate datasets, so our first step is to combine them into one and with a date set as the index. Since the earliest date of the checkin_time dataset is 01-08-2017, our new dataset will also start from that period. While the checkin_time dataset currently only contains records up to 01-12-2021, we believe the lack of data for the remainder of that month is likely due to overlapping lockdown periods. Therefore, the new dataset will extend until the end of 31-12-2021 to fully capture the impact of these lockdowns. Here is the snapshot of the new dataset. There are a total of 1,614 records and 12 columns with “usage” being the target variable.

checkin_time	usage	holiday	exam	covid	season	day	institution	gender	mean_temp	diff_temp	max_temp	min_temp
2017-08-01	148.0	Summer Vacation	0.0	0.0	Summer	Tuesday	all	both	17.8	10.2	22.7	12.5
2017-08-02	125.0	Summer Vacation	0.0	0.0	Summer	Wednesday	all	both	17.6	8.5	22.0	13.5
2017-08-03	132.0	Summer Vacation	0.0	0.0	Summer	Thursday	all	both	19.0	6.0	22.5	16.5
2017-08-04	138.0	Summer	0.0	0.0	Summer	Friday	all	both	18.3	6.4	21.8	15.4

Detail explanation of the columns is as the following table:

checkin_time	datetime	numeric	The index. Sequential dates ranging from 01-08-2017 to 31-12-2021
usage	float	numeric	The total fitness usage of that day
holiday	string	categorical	The holiday name of the day belongs to. There are 9

			types of value in total: 1. Ascension Day (40 days after Easter) 2. Christmas Break (Mid-December to early January) 3. Easter (Late March or April, varies yearly) 4. Flexible Break (RUG's flexible adjustment in accordance to national holidays) 5. Kingsday (27th of April) 6. Liberation Day (5th of May) 7. Summer Vacation (Mid July to end of August) 8. Whit (50 days after Easter) 9. None (Not a holiday)
exam	float	categorical	Whether the day is during an exam period. 1 for yes and 0 for no.
covid	float	categorical	Whether the day is during the Covid-19 lockdown period. 1 for yes and 0 for no.
season	string	categorical	The season of the date belongs to. There are 4 types of value in total: 1. Spring 2. Summer 3. Fall 4. Winter
day	string	categorical	The day of the week that the date belongs to. There are 7 types of value in total: 1. Monday 2. Tuesday 3. Wednesday 4. Thursday 5. Friday 6. Saturday 7. Sunday
institution	string	categorical	The institutions of the fitness users who went that day belong to. There are 8 types of value in total: 1. RUG 2. Hanze 3. Medewerker RUG/Hanze 4. RUG, Hanze 5. Hanze, Medewerker RUG/Hanze 6. RUG, Medewerker RUG/Hanze 7. all 8. None

gender	string	categorical	The gender of the fitness users who went that day belongs to. There are 4 types of value in total: 1. M 2. V 3. both 4. None
mean_temp	float	numeric	The mean temperature of the date, by celsius.
diff_temp	float	numeric	The difference between the minimal temperature and the maximal temperature of the date, by celsius.
max_temp	float	numeric	The maximal temperature of the date, by celsius.
min_temp	float	numeric	The minimal temperature of the date, by celsius.

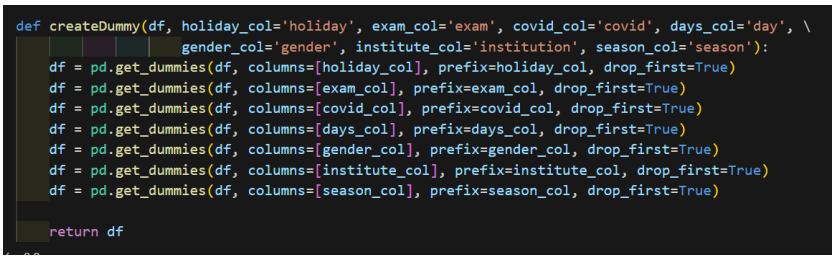
Create Dummy Variable

Dummy variables, also known as indicator variables, numeric variables that represent the different levels or categories of a categorical variable. For each category within the original categorical variable, we create a new dummy variable.

The primary reasons for us to create dummy variables for this project are the following two:

1. Many statistical models, including the regression-based models we're using, work best with numerical input.
2. If we were to directly assign numerical values to categories, the model might incorrectly assume an ordinal relationship or a consistent numerical difference between the categories, which may not be true. Dummy variables treat each category as distinct without imposing any artificial order.

In our project, we identified "holiday", "exam", "covid", "season", "day", "institution", "gender" as key categorical variables. To incorporate this information into our analysis, we created a set of dummy variables. For example, if our categorical variable was "holiday" with categories like "Summer Vacation" and "Kingsday", we would create two dummy variables: "holiday_Summer Vacation" (True if the date is during summer vacation, False otherwise) and "holiday_Kingsday" (True if the date is during kingsday, False otherwise).



```
def createDummy(df, holiday_col='holiday', exam_col='exam', covid_col='covid', days_col='day', \
                gender_col='gender', institute_col='institution', season_col='season'):
    df = pd.get_dummies(df, columns=[holiday_col], prefix=holiday_col, drop_first=True)
    df = pd.get_dummies(df, columns=[exam_col], prefix=exam_col, drop_first=True)
    df = pd.get_dummies(df, columns=[covid_col], prefix=covid_col, drop_first=True)
    df = pd.get_dummies(df, columns=[days_col], prefix=days_col, drop_first=True)
    df = pd.get_dummies(df, columns=[gender_col], prefix=gender_col, drop_first=True)
    df = pd.get_dummies(df, columns=[institute_col], prefix=institute_col, drop_first=True)
    df = pd.get_dummies(df, columns=[season_col], prefix=season_col, drop_first=True)

    return df
```

By using dummy variables, we can effectively strengthen the information contained within our categorical variables to build more comprehensive and accurate statistical models, allowing us to draw meaningful insights from our data. After performing this step, we now have 1,614 records and 32 columns with “usage” being the target variable.

Encoding

As mentioned, regression-based models work best with numerical input, our next step is to encode the boolean data from the categorical variables.

```
from sklearn.preprocessing import LabelEncoder
encoded_classes = []
encoder = LabelEncoder()

needTokenize = datas.columns[~datas.columns.isin(["usage", "mean_temp", "diff_temp", "max_temp", "min_temp"])]

for col in needTokenize:
    datas[col] = encoder.fit_transform(datas[col])
    encoded_classes.append(encoder.classes_)

datas.head(20)
```

0.0s

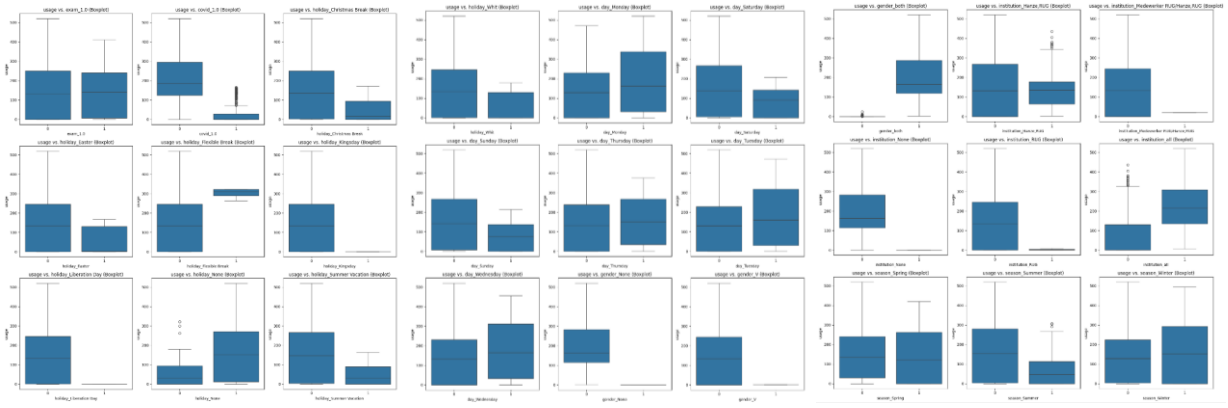
There are several encoding methods but we choose LabelEncoder from the scikit-learn library for the following reasons:

1. It is the most common and often the best practice for representing boolean features
2. It uses a single numerical column, which is memory-efficient
3. No False Ordinality: For boolean data (two categories), label encoding doesn't introduce any misleading ordinal relationships. The numerical values 0 and 1 simply represent the two distinct states.

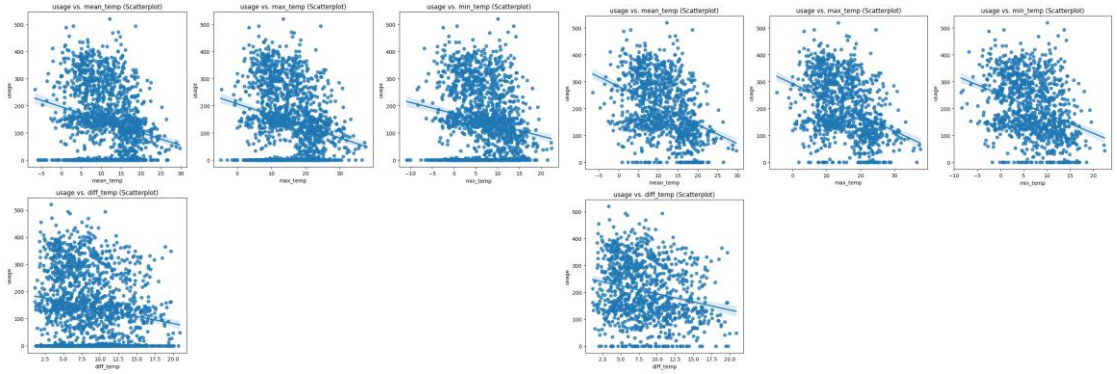
Feature Selection

Relation with target variable

Here is a visualization for the relationship between the categorical (box plot) and numeric variables (scatter plot) to the fitness usage. The box plot shows that the medians across categories are not significantly different in all cases. This could imply a weak relationship with the target variable (fitness usage).

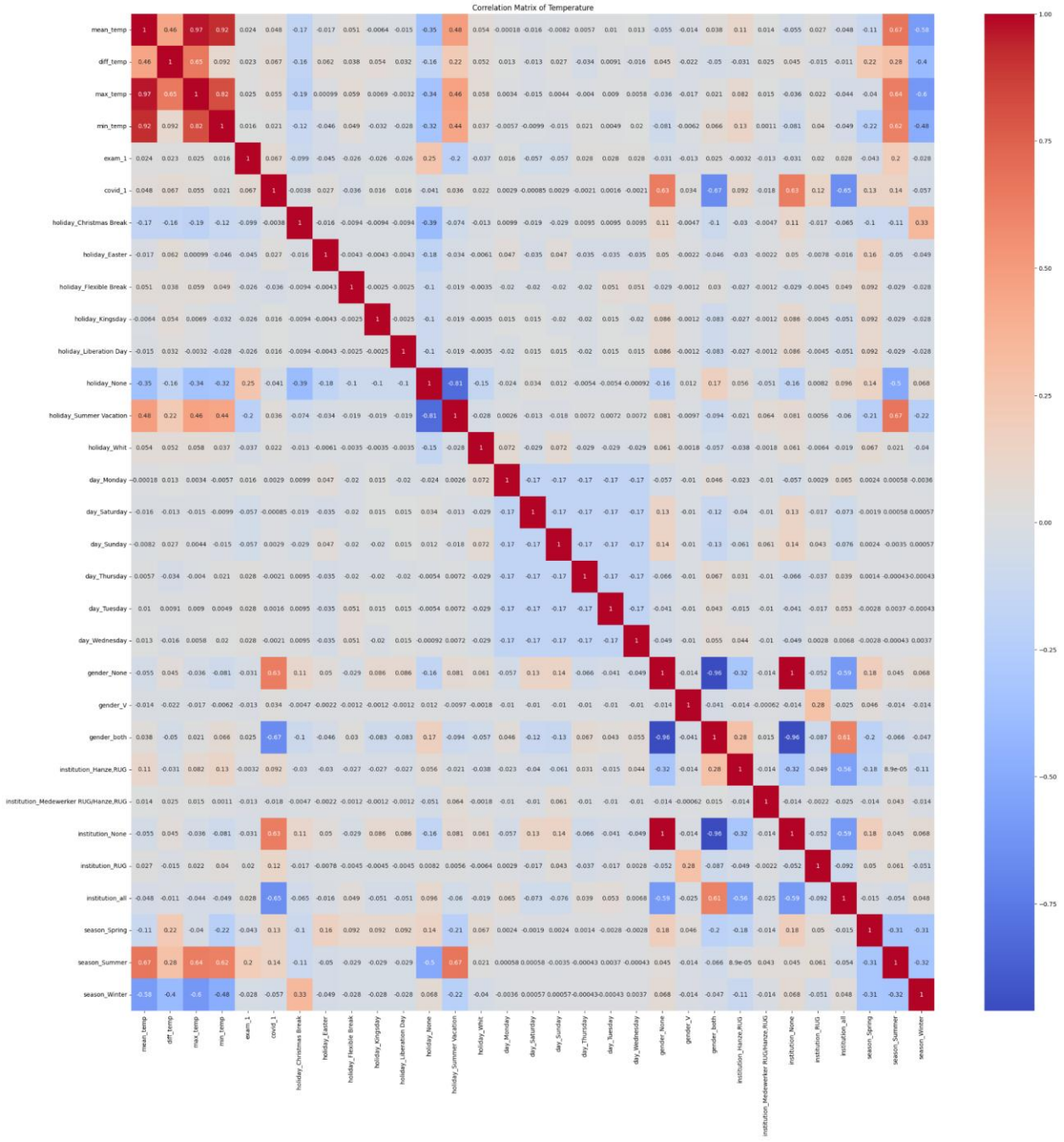


As for the numeric data (temperature), the scatter plot on the left seems to have no clear relations due to the large amount of 0 usage, there's a high chance due to the lockdowns. We filtered those out and plotted them again, the result is shown on the right chart. The regression line shows a more significant negative relationship between the variables, supporting our assumption once again.



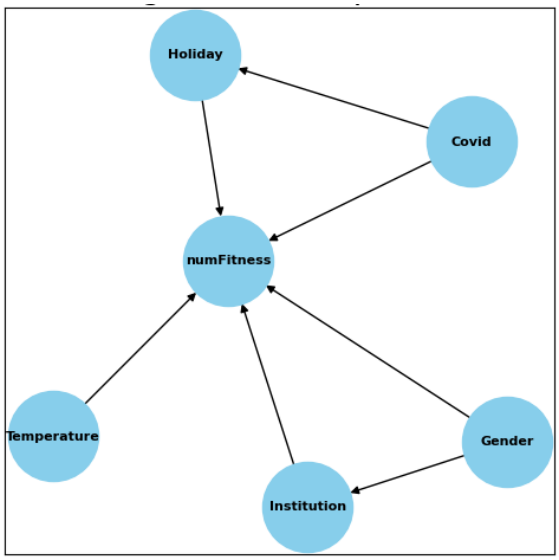
Correlation Matrix

To identify multicollinearity, we also calculated the correlation for each feature. The results are as below.



Based on the result, we can see that there are multiple features having correlation with one another. It is important to address this problem during the feature selection process by choosing either one of them to analyze.

Combining all the results from the box plot, scatter plot, and the correlation matrix, we created a DAG (Directed Acyclic Graph) with only key influencing features.



Feature Selection

There are two steps for the feature selection. First is to filter out variables that are below a certain variance threshold, since variance close to 0 indicates that the variable is tightly clustered around its mean and has the possibility of having little effect on the fitness usage. The threshold we chose for this project is 0.001. It's important to note that the optimal variance threshold can depend on the specific dataset and the nature of the features. We acknowledge that further analysis and potential experimentation with different thresholds could be considered to fine-tune the feature selection process. After the first step, we selected 29 out of the 31 features (not including the target variable).

```
from sklearn.feature_selection import VarianceThreshold

selector = VarianceThreshold(0.001)

selector.fit_transform(datas.drop("usage", axis=1))
```

✓ 0.0s

```
array([[ 0. ,  0. , 17.8, ...,  0. ,  1. ,  0. ],
       [ 0. ,  0. , 17.6, ...,  0. ,  1. ,  0. ],
       [ 0. ,  0. , 19. , ...,  0. ,  1. ,  0. ],
       ...,
       [ 0. ,  1. ,  9.5, ...,  0. ,  0. ,  1. ],
       [ 0. ,  1. , 12.9, ...,  0. ,  0. ,  1. ],
       [ 0. ,  1. , 12.8, ...,  0. ,  0. ,  1. ]], shape=(1614, 29))
```

```
# Get the indices of features with high variance
high_variance_indices = selector.get_support(indices=True)

selected_columns = ["usage"]

selected_columns.extend([predictor[i] for i in high_variance_indices.tolist()])
display(selected_columns)
```

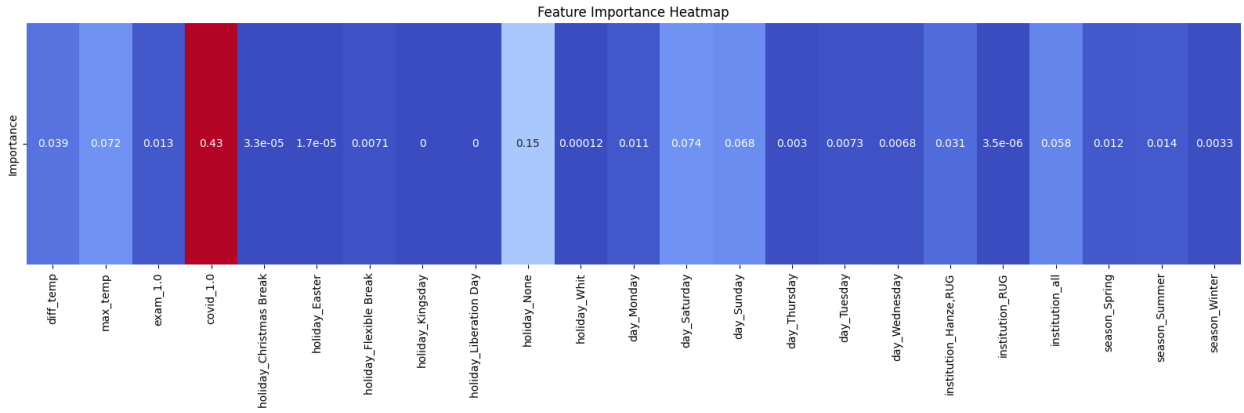
The second step is to conduct a simple random forest regression to identify which of the remaining features are actually useful for predicting the target variable (fitness usage). It ranks features by their importance in predicting the target, allowing us to select the most impactful ones for our final model and focusing on the "signal" after filtering out the "noise" of low-variance features.

Before doing so, we need to prevent multicollinearity from affecting the weight of each feature. We therefore identify feature pairs that are highly correlated within each other, remaining one feature with the highest variance from each pair. Here, only 23 features are left.

```
#prevent multilinearity
#(min_temp, max_temp, mean_temp),
#(gender_None, institution_None, institution_all)
#(gender_Both, institution_None, institution_all)
#(gender_Both, gender_None)
#(holiday_None, holiday_SummerVacation)
#(institution_None, institution_all)


selected_columns.remove("min_temp")
selected_columns.remove("mean_temp")
selected_columns.remove("gender_None")
selected_columns.remove("institution_None")
selected_columns.remove("gender_both")
selected_columns.remove("holiday_Summer Vacation")
display(selected_columns)
```

After applying the features to the random forest regression, we now have the importance of each feature. For better view, the importance is plotted as below, with warmer color representing higher importance.



We only selected features that are bigger than the median of importance. This decision is made due to models showing better results using media as a threshold.

```
# Threshold based on 75th percentile
medianThreshold = np.percentile(importance, 50) #variace 0.05|0.01; svr, decision tree
q75Threshold = np.percentile(importance, 75)
meanThreshold = np.mean(importance)
stdThreshold = np.std(importance)


selected = ["usage"]

# Select features above the threshold
selected.extend(filtered.columns.drop("usage")[importance >= medianThreshold])

display(selected)
```

Including the target variable, the selected features are shown below:

```
display(selected)
```
✓ 0.0s

```
['usage',
 'diff_temp',
 'max_temp',
 'exam_1.0',
 'covid_1.0',
 'holiday_None',
 'day_Monday',
 'day_Saturday',
 'day_Sunday',
 'institution_Hanze,RUG',
 'institution_all',
 'season_Spring',
 'season_Summer']
```

Splitting Train Test Set

Normalization

```
datas.describe()
```
✓ 0.0s

	usage	mean_temp	diff_temp	max_temp	min_temp
count	1614.000000	1614.000000	1614.000000	1614.000000	1614.000000
mean	143.255266	11.199690	8.517906	15.284634	6.766729
std	126.641825	6.074748	4.136923	7.157868	5.472484
min	0.000000	-6.600000	1.100000	-4.800000	-10.900000
25%	0.000000	6.600000	5.200000	9.700000	2.700000
50%	134.000000	11.050000	8.000000	14.850000	6.900000
75%	245.000000	16.100000	11.200000	20.900000	10.900000
max	520.000000	29.700000	20.900000	37.500000	22.400000

```
filtered.describe()
```
✓ 0.0s

	usage	diff_temp	max_temp
count	1614.000000	1614.000000	1614.000000
mean	143.255266	8.517906	15.284634
std	126.641825	4.136923	7.157868
min	0.000000	1.100000	-4.800000
25%	0.000000	5.200000	9.700000
50%	134.000000	8.000000	14.850000
75%	245.000000	11.200000	20.900000
max	520.000000	20.900000	37.500000

Before splitting into a train/test set, we need to handle the diverse range of some variables. We chose to use MinMaxScaler to scale the data to a specific range (usually between 0 and 1). Data is also separated into X (predictive variables) and Y (target variable) to prevent data leakage during the scaling. The results are shown below.

```
X_filtered.head()
```
✓ 0.0s

	0	1	2	3	4	5	6	7	8	9	10	11
0	0.459596	0.650118	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
1	0.373737	0.633570	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
2	0.247475	0.645390	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
3	0.267677	0.628842	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
4	0.439394	0.595745	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0

```
X_filtered.head()
```
✓ 0.0s

	0	1	2	3	4	5	6	7	8	9	10	11
0	0.459596	0.650118	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
1	0.373737	0.633570	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
2	0.247475	0.645390	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
3	0.267677	0.628842	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
4	0.439394	0.595745	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0

```
Y_filtered.head()
```
✓ 0.0s

	0
0	0.203846
1	0.203846
2	0.186538
3	0.194231
4	0.000000

```
Y_filtered.head()
```
✓ 0.0s

	0
0	0.203846
1	0.203846
2	0.186538
3	0.194231
4	0.000000

Splitting Dataset

We decided to apply the training and predicting to the original dataset, even after performing feature selection, mainly to establish a baseline for comparison. By running the prediction on all the initial features, we can assess the performance of a model that uses the complete set of available information. This baseline then allows us to evaluate the effectiveness of our feature selection process. So, we will be splitting both the original and the feature-selected dataset in this step.

Time Split

For time series data, it is common to split data by time sequential. This ensures that the model is trained in past data and evaluated on future, unseen data. To include part of the lockdowns data into the training, we decide to split the training data until 31-05-2021 and the test data afterwards.

```
train_end_date = datas.index.get_loc(pd.to_datetime('2021-05-31').date())
test_end_date_with_value = datas.index.get_loc(pd.to_datetime('2021-12-31').date())
```

✓ 0.0s

```
X_train = X[X.index <= train_end_date]
y_train = Y[Y.index <= train_end_date]
```

```
X_train_filtered = X_filtered[X_filtered.index <= train_end_date]
y_train_filtered = Y_filtered[Y_filtered.index <= train_end_date]
```

```
X_test = X[(X.index > train_end_date) & (X.index <= test_end_date_with_value)]
y_test = Y[(Y.index > train_end_date) & (Y.index <= test_end_date_with_value)]
```

```
X_test_filtered = X_filtered[(X_filtered.index > train_end_date) & (X_filtered.index <= test_end_date_with_value)]
y_test_filtered = Y_filtered[(Y_filtered.index > train_end_date) & (Y_filtered.index <= test_end_date_with_value)]
```

Random Split

But since the lockdowns only occurred in the last two years, we suspect the trained data may not have any information about the fact that lockdowns will significantly affect fitness usage. After consideration, we decided to also try applying train_test_split (random splitting) with 0.3 test size and check the differences with normal time series split.

```
X_tr, X_te, y_tr, y_te = train_test_split(X, Y, test_size=0.3, random_state=42)
```

✓ 0.0s

```
X_tr_f, X_te_f, y_tr_f, y_te_f = train_test_split(X_filtered, Y_filtered, test_size=0.3, random_state=42)
```

We also sorted the train/test set by time order (the index) and will apply 5-fold TimeSeriesSplit cross-validation in the training phase to prevent future data from influencing the training of past segments.

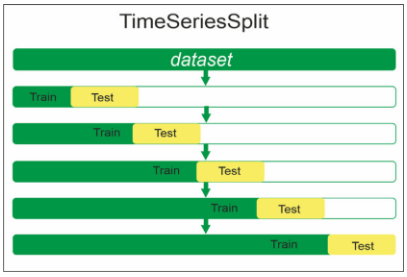
```
X_train = X_tr.sort_index()
X_test = X_te.sort_index()
```
✓ 0.0s

```
X_train_filtered = X_tr_f.sort_index()
X_test_filtered = X_te_f.sort_index()
```
✓ 0.0s

```
y_train = y_tr.sort_index()
y_test = y_te.sort_index()
```

```
y_train_filtered = y_tr_f.sort_index()
y_test_filtered = y_te_f.sort_index()
```

Demonstration of how TimeSeriesSplit cross-validation works is shown below, created by *da Silva, D. G., Geller, M. T. B., dos Santos Moura, M. S., & de Moura Meneses, A. A. (2022)*.



The shape of the train/test set is shown below.

```
print(X_train.shape)
print(X_test.shape)
```
✓ 0.0s
```
(1129, 31)
(485, 31)
```

```
print(X_train_filtered.shape)
print(X_test_filtered.shape)
```
✓ 0.0s
```
(1129, 12)
(485, 12)
```

```
print(y_train.shape)
print(y_test.shape)
```
✓ 0.0s
```
(1129, 1)
(485, 1)
```

```
print(y_train_filtered.shape)
print(y_test_filtered.shape)
```
✓ 0.0s
```
(1129, 1)
(485, 1)
```

Algorithm Candidates

For the algorithm models, we decided to do a regression approach since our dataset consists of continuous data. The primary goal of a regression algorithm is to create a best-fit line or curve that captures the relationship within the data. To evaluate how well the model performs, three key metrics are commonly considered: bias, variance, and overall prediction error. In order for us to get the most optimal accuracy, we decided to test out several models. These models include Logistic Regression, Linear Regression, Random Forest, XG Boost, SVR (Support Vector Regression), Poisson Regression, and Decision Tree. We will perform the models on both original training data and training data with feature selection.

As mentioned above, we used two methods to split our data (Random Split and Time Split). We then applied our models to the corresponding training and testing sets generated by each method.

Logistic Regression

The first algorithm we tried using for our project was logistic regression. Despite the name, logistic regression is actually a supervised machine learning method that is used for classification tasks. It is suitable when the output target is categorical (usually binary), therefore this model cannot be used for our project.

```
logModel = LogisticRegression(max_iter=200)
logModel.fit(X_train, y_train)

# predict the target value using the train data
y_pred = logModel.predict(X_test)

print_model_score('Logistic Regression (original data)', y_test, y_pred, o_features)
#print(f'Accuracy: {accuracy_score(y_test, y_pred)}')
#print('Confusion Matrix:')
#print(confusion_matrix(y_test, y_pred))
#plt.subplots(figsize=(8,5))
#sns.heatmap(confusion_matrix(y_test, y_pred), annot=True)
#plt.xlabel("y_head")
#plt.ylabel("y_true")
#plt.show()
#print('Classification Report:')
#print(classification_report(y_test, y_pred))
```

We still tried to code the algorithm with our data, but we cannot successfully run it since our data is continuous and like we mentioned before, the model only accepts categorical data.

```
...
    224         "classifier, which expects discrete classes on a "
    225         "regression target with continuous values."
    226     )

ValueError: Unknown label type: continuous. Maybe you are trying to fit a classifier, which expects discrete classes
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Linear Regression

The second algorithm we tried for our project was linear regression. Linear regression is a supervised machine learning algorithm that works by fitting a straight line through the data that best represents the relationship between the input features and the target variable. It is used for continuous numerical data. Since our project involved predicting the number of fitness visitors, linear regression was appropriate for the task. We tried using the model on both the original train dataset and selected features dataset.

Random split

Original train dataset:

```
  model = LinearRegression()
  model.fit(X_train, y_train)

  LinearRegressionModel = model.predict(X_test)
  print_model_score('Linear Regession', y_test, LinearRegressionModel, o_features)
```

```
Linear Regession
Mean Squared Error (MSE): 0.015820435361865354
Mean Absolute Error (MAE): 0.09928206349698274
R-squared (R2): 0.7484793883492233
Adjusted R-squared (Adjusted-R2): 0.7312671610618633
```

Train data with feature selection:

```
model_filtered = LinearRegression()
model_filtered.fit(X_train_filtered, y_train_filtered)

LinearRegressionModelFiltered = model_filtered.predict(X_test_filtered)
print_model_score('Linear Regession', y_test_filtered, LinearRegressionModelFiltered, f_features)
```

```
Linear Regession
Mean Squared Error (MSE): 0.016262220129365183
Mean Absolute Error (MAE): 0.0996866314033535
R-squared (R2): 0.7414556894181941
Adjusted R-squared (Adjusted-R2): 0.7348825289796737
```

Based on the R squared above, we can tell that the linear regression has obtained a decent percentage of accuracy, though the training dataset with feature selection has a higher accuracy than the original training dataset since it has less features and is more correlated with the target value. However, the simplicity of the model limited its performance when dealing with more complex relationships in the data.

Time split

Original Train dataset:

```
model = LinearRegression()
model.fit(X_train, y_train)

LinearRegressionModel = model.predict(X_test)
print_model_score('Linear Regession', y_test, LinearRegressionModel, o_features)
```

```
Linear Regession
Mean Squared Error (MSE): 0.026294955812462367
Mean Absolute Error (MAE): 0.1425740765966117
R-squared (R2): -0.1621262934239649
Adjusted R-squared (Adjusted-R2): -0.3600708818643106
```

Train data with feature selection:

```
model_filtered = LinearRegression()
model_filtered.fit(X_train_filtered, y_train_filtered)

LinearRegressionModelFiltered = model_filtered.predict(X_test_filtered)
print_model_score('Linear Regession', y_test_filtered, LinearRegressionModelFiltered, f_features)
```

```
Linear Regession
Mean Squared Error (MSE): 0.022529681555185407
Mean Absolute Error (MAE): 0.12830973600346463
R-squared (R2): 0.004282969536655634
Adjusted R-squared (Adjusted-R2): -0.05516282332682754
```

Random Forest

Next, we decided to implement the random forest model into our project. Random Forest is a supervised learning algorithm that creates multiple decision trees during training and combines their outputs, which helps to reduce overfitting and improve generalization. It was chosen for this task due to its ability to handle complex relationships, robustness to outliers, and strong performance even without extensive hyperparameter tuning.

Random split

Original train dataset:

```
rf_model = RandomForestRegressor()
rf_model.fit(X_train, y_train)

RandomForestModel = rf_model.predict(X_test)
print_model_score('Random Forest', y_test, RandomForestModel, o_features)
```

```
Mean Squared Error (MSE): 0.008226962595885732
Mean Absolute Error (MAE): 0.05709633200554299
R-squared (R2): 0.8692039367555522
Adjusted R-squared (Adjusted-R2): 0.8602532127807666
```

Train data with feature selection:

```
rf_model_filtered = RandomForestRegressor()
rf_model_filtered.fit(X_train_filtered, y_train_filtered)

RandomForestModelFiltered = rf_model_filtered.predict(X_test_filtered)
print_model_score('Random Forest Filtered', y_test_filtered, RandomForestModelFiltered, f_features)
```

```
Random Forest Filtered
Mean Squared Error (MSE): 0.009029203060291655
Mean Absolute Error (MAE): 0.060831245043616186
R-squared (R2): 0.8564495461409446
Adjusted R-squared (Adjusted-R2): 0.8527999583309687
```

Time split

Original train dataset:

```
rf_model = RandomForestRegressor()
rf_model.fit(X_train, y_train)

RandomForestModel = rf_model.predict(X_test)
print_model_score('Random Forest', y_test, RandomForestModel, o_features)
```

```
Mean Squared Error (MSE): 0.005614630790311547
Mean Absolute Error (MAE): 0.0504392935049984
R-squared (R2): 0.75185696770799
Adjusted R-squared (Adjusted-R2): 0.7095908468230873
```

Train dataset with feature selection:

```
rf_model_filtered = RandomForestRegressor()
rf_model_filtered.fit(X_train_filtered, y_train_filtered)

RandomForestModelFiltered = rf_model_filtered.predict(X_test_filtered)
print_model_score('Random Forest Filtered', y_test_filtered, RandomForestModelFiltered, f_features)
```

```
Mean Squared Error (MSE): 0.006663132335361943
Mean Absolute Error (MAE): 0.053845513892242895
R-squared (R2): 0.7055176156706986
Adjusted R-squared (Adjusted-R2): 0.6879365778002926
```

XG Boost

The fourth algorithm we decided to use is XG Boost (Extreme Gradient Boosting), a supervised learning algorithm that is suitable for both regression and classification tasks. XGBoost builds multiple iterations of decision trees, but instead of building them all at once (like Random Forest), it builds them one at a time, and each new tree tries to fix the errors made by the previous ones.

For this model, we added a set parameter function for it to choose its best parameters to avoid overfitting. Overfitting occurs when a model learns the training data too well, including the noise and outliers, and performs poorly on new, unseen data.

Random split

Original train dataset:

```
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set parameters for the XGBoost model
params = {
    'objective': 'multi:softmax',  # for multi-class classification
    'num_class': 3,                # number of classes in Iris dataset
    'max_depth': 4,                # depth of the trees
    'eta': 0.1,                    # learning rate
    'eval_metric': 'merror'        # evaluation metric for multi-class classification
}

# Train the model using the training data
num_round = 100  # Number of boosting rounds
bst = xgb.train(params, dtrain, num_round)

# Make predictions on the test set
XGBoostModel = bst.predict(dtest)

print_model_score('XG Model', y_test, XGBoostModel, o_features)
```

```
XG Model
Mean Squared Error (MSE): 0.1467624455690384
Mean Absolute Error (MAE): 0.28959161043167114
R-squared (R2): -1.3332972526550293
Adjusted R-squared (Adjusted-R2): -1.492971016081753
```

Train data with feature selection:

```
dtrain_filtered = xgb.DMatrix(X_train, label=y_train_filtered)
dtest_filtered = xgb.DMatrix(X_test, label=y_test_filtered)

# Set parameters for the XGBoost model
params = {
    'objective': 'multi:softmax',  # for multi-class classification
    'num_class': 3,                # number of classes in Iris dataset
    'max_depth': 4,                # depth of the trees
    'eta': 0.1,                    # learning rate
    'eval_metric': 'merror'        # evaluation metric for multi-class classification
}

# Train the model using the training data
num_round = 100  # Number of boosting rounds
bst_filtered = xgb.train(params, dtrain_filtered, num_round)

# Make predictions on the test set
XGBoostModel_filtered = bst_filtered.predict(dtest_filtered)

print_model_score('XG Model', y_test_filtered, XGBoostModel_filtered, f_features)
```

```
XG Model
Mean Squared Error (MSE): 0.1467624455690384
Mean Absolute Error (MAE): 0.28959161043167114
R-squared (R2): -1.3332972526550293
Adjusted R-squared (Adjusted-R2): -1.3926183692479537
```

Time split

Original train dataset:

```
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set parameters for the XGBoost model
params = {
    'objective': 'multi:softmax',  # for multi-class classification
    'num_class': 3,                # number of classes in Iris dataset
    'max_depth': 4,                # depth of the trees
    'eta': 0.1,                    # learning rate
    'eval_metric': 'merror'        # evaluation metric for multi-class classification
}

# Train the model using the training data
num_round = 100  # Number of boosting rounds
bst = xgb.train(params, dtrain, num_round)

# Make predictions on the test set
XGBoostModel = bst.predict(dtest)

print_model_score('XG Model', y_test, XGBoostModel, o_features)
```

```
XG Model
Mean Squared Error (MSE): 0.41773852705955505
Mean Absolute Error (MAE): 0.5284597277641296
R-squared (R2): -17.462282180786133
Adjusted R-squared (Adjusted-R2): -20.60695661817278
```

Train dataset with feature selection:

```
dtrain_filtered = xgb.DMatrix(X_train, label=y_train_filtered)
dtest_filtered = xgb.DMatrix(X_test, label=y_test_filtered)

# Set parameters for the XGBoost model
params = {
    'objective': 'multi:softmax',  # for multi-class classification
    'num_class': 3,                # number of classes in Iris dataset
    'max_depth': 4,                # depth of the trees
    'eta': 0.1,                    # learning rate
    'eval_metric': 'merror'        # evaluation metric for multi-class classification
}

# Train the model using the training data
num_round = 100  # Number of boosting rounds
bst_filtered = xgb.train(params, dtrain_filtered, num_round)

# Make predictions on the test set
XGBoostModel_filtered = bst_filtered.predict(dtest_filtered)

print_model_score('XG Model', y_test_filtered, XGBoostModel_filtered, f_features)
```

```
XG Model
Mean Squared Error (MSE): 0.41773852705955505
Mean Absolute Error (MAE): 0.5284597277641296
R-squared (R2): -17.462282180786133
Adjusted R-squared (Adjusted-R2): -18.56450798262411
```

SVR

The next algorithm we tested on our dataset is Support Vector Regression. SVR is a supervised learning algorithm that has the same basic principles as Support Vector Machine (typically used for classification) but has been adapted for regression tasks. Unlike other linear regressions models, SVR will try to fit the best possible line within a margin of tolerance (epsilon), which means it does not care about small errors as long as they are still in that margin. It uses key data points or support vectors to define the margin.

Random Split

Original train dataset:
Before we trained the model, we defined a function to help choose the best parameters for the model to avoid overfitting.

```python
# Create and train the SVR model
param_distributions = {
    'C': uniform(1, 100),
    'epsilon': uniform(0.01, 0.2),
    'kernel': ['rbf', 'linear', 'poly'],
    'gamma': ['scale', 'auto'],
    'degree': randint(2, 5) #integer values between 2 and 4. Only used with poly kernel.
}

# Perform grid search
random_search = RandomizedSearchCV(
    SVR(),
    param_distributions,
    n_iter=100,  # Number of parameter combinations to try
    cv=tscv,  # Cross-validation folds
    scoring='neg_mean_squared_error',
    random_state=42,
    n_jobs=-1 #use all available cores.
)

random_search.fit(X_train, y_train[0])

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

```
# Use the best model
svrModel = random_search.best_estimator_
svrModel.fit(X_train, y_train[0])

# Predictions
predictSVR = svrModel.predict(X_test)
print_model_score('SVR (original data)', y_test, predictSVR, o_features)
```

```
SVR (original data)
Mean Squared Error (MSE): 0.008850274152268084
Mean Absolute Error (MAE): 0.06495832353795412
R-squared (R2): 0.8592942408259325
Adjusted R-squared (Adjusted-R2): 0.8496653698890758
```

Train dataset with feature selection:

```
random_search.fit(X_train_filtered, y_train_filtered[0])

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'min_samples_split': 10, 'min_samples_leaf': 4, 'max_features': 'sqrt', 'max_depth': 7}
```

```
svrModel_filtered = random_search.best_estimator_
svrModel_filtered.fit(X_train_filtered, y_train_filtered[0])

# Predictions
predictSVR_filtered = svrModel_filtered.predict(X_test_filtered)
print_model_score('SVR (feature selected)', y_test_filtered, predictSVR_filtered, f_features)
```

```
SVR (feature selected)
Mean Squared Error (MSE): 0.011340351758450832
Mean Absolute Error (MAE): 0.06750171418709487
R-squared (R2): 0.819705833285981
Adjusted R-squared (Adjusted-R2): 0.8151220832847771
```

Time split

Original train dataset:

```python
# Create and train the SVR model
param_distributions = {
    'C': uniform(1, 100),
    'epsilon': uniform(0.01, 0.2),
    'kernel': ['rbf', 'linear', 'poly'],
    'gamma': ['scale', 'auto'],
    'degree': randint(2, 5) #integer values between 2 and 4. Only used with poly kernel.
}

# Perform grid search
random_search = RandomizedSearchCV(
    SVR(),
    param_distributions,
    n_iter=100,  # Number of parameter combinations to try
    cv=tscv,  # Cross-validation folds
    scoring='neg_mean_squared_error',
    random_state=42,
    n_jobs=-1 #use all available cores.
)

random_search.fit(X_train, y_train[0])

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

Python

```
Best Parameters: {'C': np.float64(12.816482762165625), 'degree': 2, 'epsilon': np.float64(0.07966732089063946), 'gamma': 'scal
```

```python
# Use the best model
svrModel = random_search.best_estimator_
svrModel.fit(X_train, y_train[0])

# Predictions
predictSVR = svrModel.predict(X_test)
print_model_score('SVR (original data)', y_test, predictSVR, o_features)
```

```
SVR (original data)
Mean Squared Error (MSE): 0.013142027307495162
Mean Absolute Error (MAE): 0.08857874507332564
R-squared (R2): 0.41917774679440756
Adjusted R-squared (Adjusted-R2): 0.3202464838857627
```

Train dataset with feature selection:

```python
random_search.fit(X_train_filtered, y_train_filtered[0])

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'min_samples_split': 5, 'min_samples_leaf': 4, 'max_features': None, 'max_depth': 7}
```

```python
svrModel_filtered = random_search.best_estimator_
svrModel_filtered.fit(X_train_filtered, y_train_filtered[0])

# Predictions
predictSVR_filtered = svrModel_filtered.predict(X_test_filtered)
print_model_score('SVR (feature selected)', y_test_filtered, predictSVR_filtered, f_features)
```

```
Mean Squared Error (MSE): 0.011317187381858091
Mean Absolute Error (MAE): 0.06955138286107218
R-squared (R2): 0.4998279853419688
Adjusted R-squared (Adjusted-R2): 0.4699669695414894
```

Poisson Regression

We also decided to test the Poisson Regression model on our dataset, since it is a supervised regression algorithm that's usually used to predict count-based outcomes such as number of visitors, number of purchases, etc. This model assumes that the target variable (y) follows a poisson distribution, which is typically used to model the number of times an event happens in a fixed time or space.

Random Split

Original train dataset:
We also defined the best parameter function on this model to avoid overfitting.

```
def poisson(time_series_splits):
    best_mse = float('inf')
    best_model = None
    for X_train_fold, y_train_fold, X_val_fold, y_val_fold in time_series_splits:
        # Fit the Poisson GLM model
        poissonModel = sm.GLM(y_train_fold, X_train_fold, family=sm.families.Poisson()).fit()

        # Make predictions on the validation fold
        y_pred = poissonModel.predict(X_val_fold)

        # Evaluate the model (using MSE)
        mse = mean_squared_error(y_val_fold, y_pred)

        print(f"Validation Fold MSE: {mse}")

        # Check if this model is the best so far
        if mse < best_mse:
            best_mse = mse
            best_model = poissonModel
    return best_model
```

Before we train the model, we added a time series validation fold to make sure the model will actually work for future data, not just the training data. It will give us a more honest look at how the model will perform overtime.

```
# Create time series validation folds
poissonSplit = time_series_validation_split(X_train, y_train, tscv)

best_model = poisson(poissonSplit)
```

```
Validation Fold MSE: 0.30737270017250623
Validation Fold MSE: 0.007166009592938658
Validation Fold MSE: 0.005949355571327978
Validation Fold MSE: 0.005040188621448428
Validation Fold MSE: 0.010113214531250086
```

```
y_test_pred = best_model.predict(X_test)
print_model_score("Poisson Regression (Original Data)", y_test, y_test_pred, o_features)
```

```
Poisson Regression (Original Data)
Mean Squared Error (MSE): 0.008384936883106332
Mean Absolute Error (MAE): 0.056964922203507684
R-squared (R2): 0.86669238834124
Adjusted R-squared (Adjusted-R2): 0.857569792399912
```

Train dataset with feature selection:

```
# Create time series validation folds
poissonSplit_filtered = time_series_validation_split(X_train_filtered, y_train_filtered, tscv)

best_model_filtered = poisson(poissonSplit_filtered)
```

```
Validation Fold MSE: 0.08166617462434136
Validation Fold MSE: 0.011924881412740793
Validation Fold MSE: 0.2901428331263704
Validation Fold MSE: 0.02104924245019485
Validation Fold MSE: 0.02465774418330508
```

```
# Predictions
y_test_pred = best_model_filtered.predict(X_test_filtered)
print_model_score("Poisson Regression (Feature Selected)", y_test_filtered, y_test_pred, f_features)
```

```
Poisson Regression (Feature Selected)
Mean Squared Error (MSE): 0.18735472396272576
Mean Absolute Error (MAE): 0.25350229894232196
R-squared (R2): -1.978652210820762
Adjusted R-squared (Adjusted-R2): -2.0543806568585783
```

Time Split

Original train dataset:

```
def poisson(time_series_splits):
    best_mse = float('inf')
    best_model = None
    for X_train_fold, y_train_fold, X_val_fold, y_val_fold in time_series_splits:
        # Fit the Poisson GLM model
        poissonModel = sm.GLM(y_train_fold, X_train_fold, family=sm.families.Poisson()).fit()

        # Make predictions on the validation fold
        y_pred = poissonModel.predict(X_val_fold)

        # Evaluate the model (using MSE)
        mse = mean_squared_error(y_val_fold, y_pred)

        print(f"Validation Fold MSE: {mse}")

        # Check if this model is the best so far
        if mse < best_mse:
            best_mse = mse
            best_model = poissonModel
    return best_model
```

```
# Create time series validation folds
poissonSplit = time_series_validation_split(X_train, y_train, tscv)

best_model = poisson(poissonSplit)
```

```
Validation Fold MSE: 0.649444185241198
Validation Fold MSE: 0.009006657843444526
Validation Fold MSE: 0.010705389152477911
Validation Fold MSE: 0.013468778443901444
Validation Fold MSE: 0.004083337780099401
```

```
y_test_pred = best_model.predict(X_test)
print_model_score("Poisson Regression (Original Data)", y_test, y_test_pred, o_features)
```

```
Poisson Regression (Original Data)
Mean Squared Error (MSE): 0.011608492620377645
Mean Absolute Error (MAE): 0.07046781005042725
R-squared (R2): 0.4869535207674641
Adjusted R-squared (Adjusted-R2): 0.3995664830959883
```

Train dataset with feature selection:

```
# Create time series validation folds
poissonSplit_filtered = time_series_validation_split(X_train_filtered, y_train_filtered, tscv)

best_model_filtered = poisson(poissonSplit_filtered)
```

```
Validation Fold MSE: 0.07147125476516755
Validation Fold MSE: 0.025263825088650195
Validation Fold MSE: 0.02071994497997215
Validation Fold MSE: 0.3201777966582165
Validation Fold MSE: 0.014590559341301665
```

```
# Predictions
y_test_pred = best_model_filtered.predict(X_test_filtered)
print_model_score("Poisson Regression (Feature Selected)", y_test_filtered, y_test_pred, f_features)
```

```
Poisson Regression (Feature Selected)
Mean Squared Error (MSE): 0.02941279746015649
Mean Absolute Error (MAE): 0.11755463096857038
R-squared (R2): -0.2999217620058279
Adjusted R-squared (Adjusted-R2): -0.37752903137931004
```

Decision Tree

Lastly, we decided to experiment with Decision Tree since it is a popular supervised machine learning for regression tasks. The model works by learning simple decision rules from the data features to make predictions. Decision Tree is the simpler version and is often used as building blocks for random forests.

Random Split

Original Train Dataset:

Again, we defined the best parameter feature to help the model avoid overfitting.

```
# Define the parameter grid
param_distributions = {
    'max_depth': [3, 5, 7, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None]
}

# Create the RandomizedSearchCV object
random_search = RandomizedSearchCV(
    DecisionTreeRegressor(random_state=42),
    param_distributions,
    n_iter=100,
    cv=tscv,
    scoring='neg_mean_squared_error',
    random_state=42,
    error_score='raise')

# Fit the grid search to the data
random_search.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'min_samples_split': 10, 'min_samples_leaf': 1, 'max_features': 'sqrt', 'max_depth': 7}
```

```
# Create the Decision Tree Regression model
treeModel = random_search.best_estimator_
treeModel.fit(X_train, y_train)

# Make predictions on the test set
predictTree = treeModel.predict(X_test)
print_model_score('Decision Tree Regression (Original Data)', y_test, predictTree, o_features)
```

```
Decision Tree Regression (Original Data)
Mean Squared Error (MSE): 0.018032275516855153
Mean Absolute Error (MAE): 0.087396066672263
R-squared (R2): 0.7133145287273586
Adjusted R-squared (Adjusted-R2): 0.6936958761678622
```

Train dataset with feature selection:

```
# Fit the grid search to the data
random_search.fit(X_train_filtered, y_train_filtered)

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'min_samples_split': 10, 'min_samples_leaf': 4, 'max_features': 'sqrt', 'max_depth': 7}
```

```
treeModel_filtered = random_search.best_estimator_
treeModel_filtered.fit(X_train_filtered, y_train_filtered)

# Make predictions on the test set
predictTree_filtered = treeModel_filtered.predict(X_test_filtered)
print_model_score('Decision Tree Regression (Original Data)', y_test_filtered, predictTree_filtered, f_features)
```

```
Decision Tree Regression (Original Data)
Mean Squared Error (MSE): 0.011340351758450832
Mean Absolute Error (MAE): 0.06750171418709487
R-squared (R2): 0.819705833285981
Adjusted R-squared (Adjusted-R2): 0.8151220832847771
```

Time Split

Original train dataset:

```
# Define the parameter grid
param_distributions = {
    'max_depth': [3, 5, 7, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None]
}

# Create the RandomizedSearchCV object
random_search = RandomizedSearchCV(
    DecisionTreeRegressor(random_state=42),
    param_distributions,
    n_iter=100,
    cv=tscv,
    scoring='neg_mean_squared_error',
    random_state=42,
    error_score='raise')

# Fit the grid search to the data
random_search.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'min_samples_split': 10, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': 7}
```

```
# Create the Decision Tree Regression model
treeModel = random_search.best_estimator_
treeModel.fit(X_train, y_train)

# Make predictions on the test set
predictTree = treeModel.predict(X_test)
print_model_score('Decision Tree Regression (Original Data)', y_test, predictTree, o_features)
```

```
Decision Tree Regression (Original Data)
Mean Squared Error (MSE): 0.016661920095010954
Mean Absolute Error (MAE): 0.0818880204698651
R-squared (R2): 0.2636133112585709
Adjusted R-squared (Adjusted-R2): 0.1381848093300857
```

Train dataset with feature selection:

```
# Fit the grid search to the data
random_search.fit(X_train_filtered, y_train_filtered)

# Print the best parameters
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'min_samples_split': 5, 'min_samples_leaf': 4, 'max_features': None, 'max_depth': 7}
```

```
treeModel_filtered = random_search.best_estimator_
treeModel_filtered.fit(X_train_filtered, y_train_filtered)

# Make predictions on the test set
predictTree_filtered = treeModel_filtered.predict(X_test_filtered)
print_model_score('Decision Tree Regression (Original Data)', y_test_filtered, predictTree_filtered, f_features)
```

```
Decision Tree Regression (Original Data)
Mean Squared Error (MSE): 0.02258581791606495
Mean Absolute Error (MAE): 0.09667347716843014
R-squared (R2): 0.0018019788302808815
Adjusted R-squared (Adjusted-R2): -0.0577919328813441
```

Best Performed Algorithm

Based on all the models we applied to our dataset; we have compiled an accuracy summary in the table below.

Algorithm Model	Random Split		Time Split	
	Original Train Dataset	Train Dataset with Feature Selection	Original Train Dataset	Train Dataset with Feature Selection
Linear Regression	73.13%	73.49%	36.01%	-5.52%
Random Forest	86.03%	85.36%	70.96%	68.79%
XG Boost	-149.3%	-139.26%	-2060.69%	-1856.45%
SVR	84.97%	81.51%	32.02%	46.99%
Poisson Regression	85.76%	-205.44%	39.96%	-37.75%
Decision Tree	69.37%	81.51%	13.82%	-5.78%

From the table above we can conclude that:

Best Random Split Performers:

- Random Forest showed the highest accuracy using both the original training dataset (86.02%) and after feature selection (85.36%).

Best Time Split Performers:

- Random Forest in this case also showed the highest accuracy using both the original training dataset (70.96%) and after feature selection (68.79%).

Worst Performer (Across Both Splits):

- XGBoost consistently underperformed across all scenarios, showing negative accuracy values, especially under Time Split (-2060.69%, -1856.45%), indicating inappropriate model choice.

Time Split vs. Random Split:

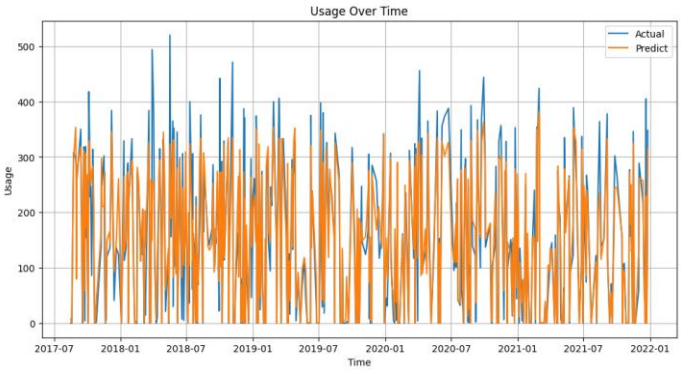
- All models generally performed **worse on Time Split**, highlighting the challenge of forecasting time dependent data.
- For example, Linear Regression dropped from 73.13% (Random) to 36.01% (Time), and even further to -5.52% after feature selection.

Impact of Feature Selection:

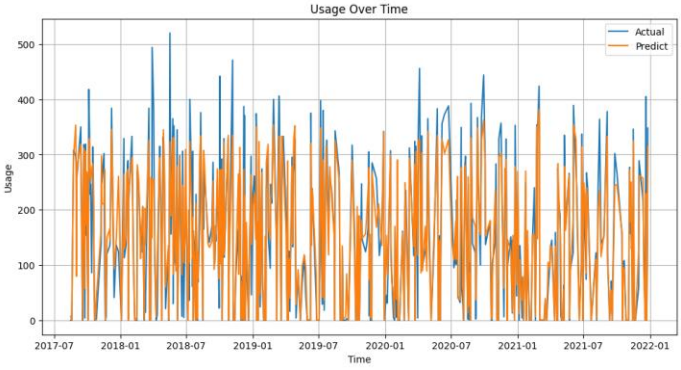
- Feature selection had a positive impact for some models (e.g. Decision Tree improved from 69.37% to 81.51%), but caused significant drops in others, such as Poisson Regression, which fell to -205.44% under Random Split.

After analyzing the models and their accuracies, we generated multiple line plots using the best-performing models to visualize their predictions.

Random Forest (Random split, Original train dataset)

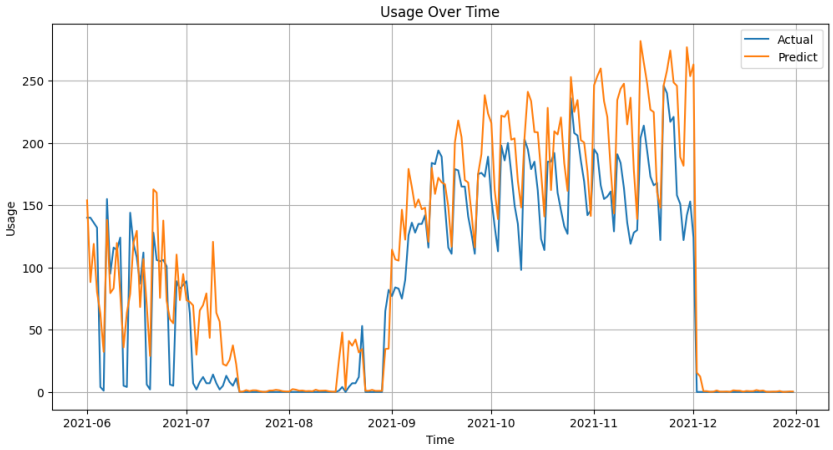


Random Forest (Random split, Feature selected train dataset)

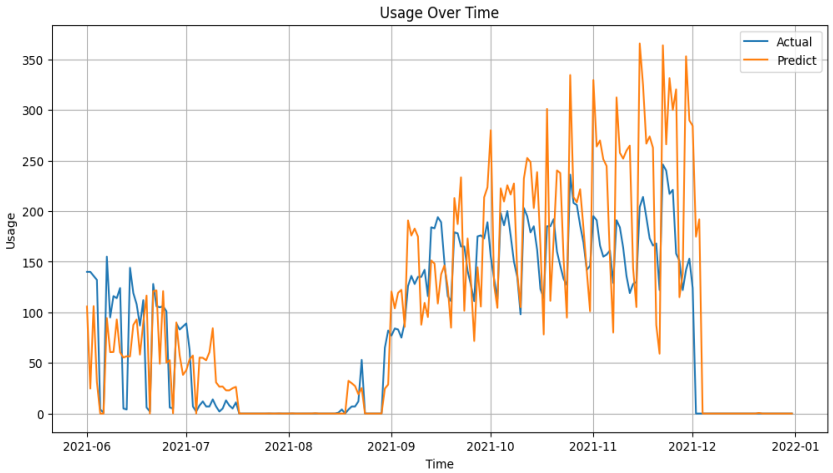


Both models, using random split techniques, show predicted values (orange) that roughly follow the trend of actual usage (blue), although with noticeable fluctuations and noise.

Random Forest (Time split, Original train dataset)



Random Forest (Time split, Feature selected train dataset)



Both models generally capture the overall trend, however, they tend to overpredict during peak periods, especially noticeable in the later months of 2021.