Frankfurt School of Finance & Management

German Excellence. Global Relevance.

Introduction to Data Analytics in Business

Bikerus

Predicting bike rentals

Yannik Suhre
Skyler MacGowan
Jacob Umland
Jan Faulstich
Sebastian Sydow

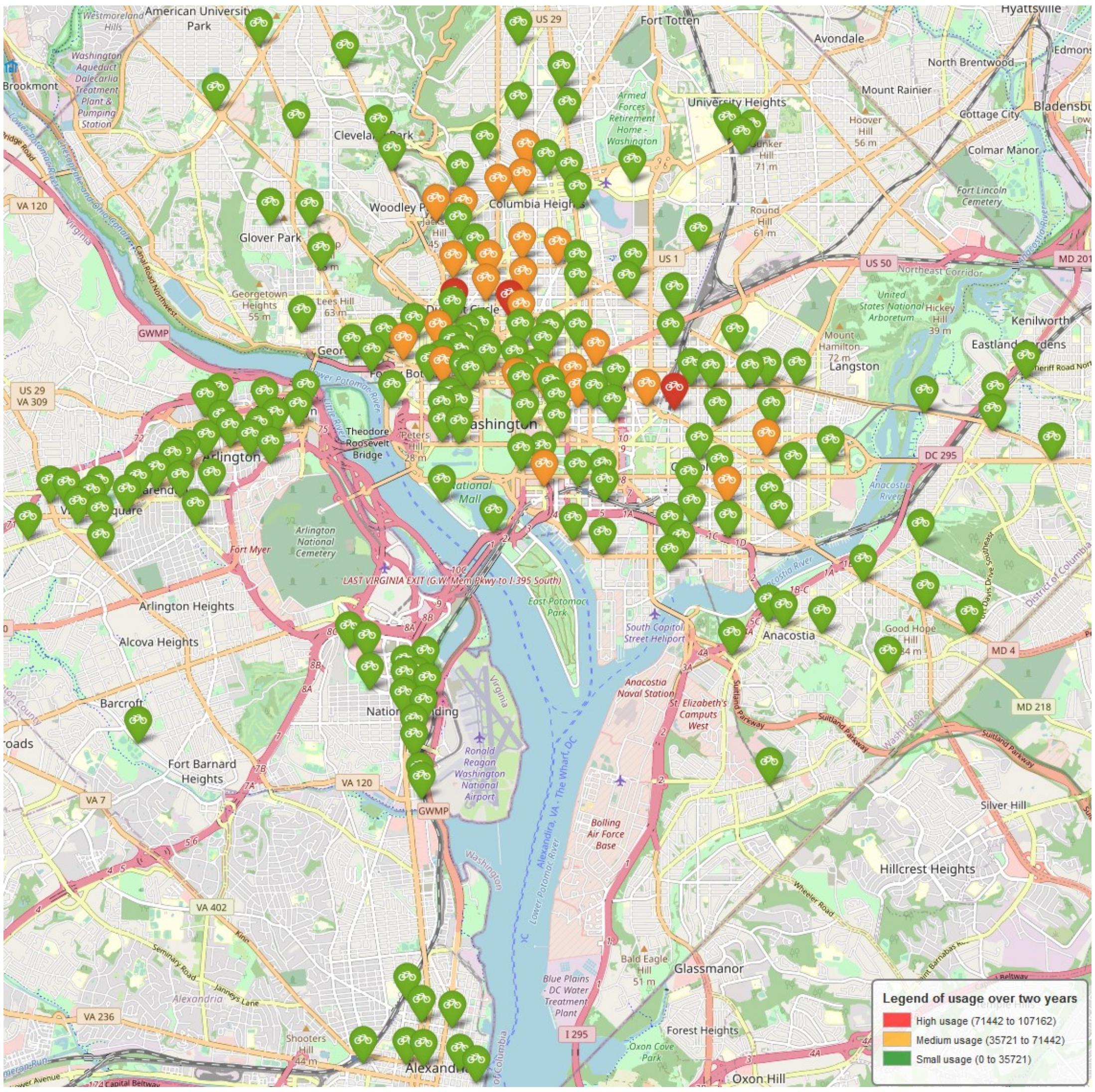
January 1, 2021

Contents

1 Introduction

1.1 The Project

The objective of the *Bike Sharing and Rental Demand Estimation* project is to build a predictive model able to forecast the number of bike rentals within a specified hour-long time frame in Washington D.C. As the bike sharing rental process is highly correlated to environmental and seasonal settings, most of the data provided relates to these factors (e.g. temperature, wind speed, day of week, time of day). The data is broken down into hourly segments for every day during 2011 and 2012. Figure 1 below provides a visualization of the degree of usage of each bike sharing and rental station in Washington D.C. during this time frame.



Legend of usage over two years
🟥 High usage (71442 to 107162)
🟧 Medium usage (35721 to 71442)
🟩 Small usage (0 to 35721)

Figure 1: Visualization of bike stations in 2011-2012 and their usage during these years

1.2 Context

1.2.1 What is Bike Sharing?

Broadly, bike sharing is a service wherein individuals are able to make use of a bike they do not own for a short period of time, typically less than 30 minutes. These arrangements require participants to pay a small fee to rent the bike, with operators also financing the service through a variety of means including sponsorship agreements, harvesting user-data, and government subsidies. The implementation of bike sharing systems is sometimes done in collaboration with the public sector, though the actual operation and ownership thereof is conducted by private companies. The private interest in this market is perhaps best manifested by Lyft's 2018 acquisition of Motivate (the operator of Capital Bikeshare and New York's Citi Bike) for approximately $250 million [1].[1]

1.2.2 A Historical Perspective

The first well-known bike sharing initiative was started in Amsterdam in 1965 by Luud Schimmelpennik [2]. This was known as the "Witte Fietsenplan" (White Bicycle Plan) and the idea was to make a number of bicycles available to the public to use anywhere in the city free of charge [2]. Within less than a month however it was clear this plan would not succeed as less than half the initial number of bikes remained due to a combination of thefts and police confiscations (this initiative did not have political support) [3].

Much more successful and well ahead of its time was Schimmelpennik's "Witkar" initiative, which operated for approximately 10 years in the 1970s and early 1980s [2]. Essentially, this involved 25 specially-designed *witkaren* that users could rent to drive from one parking station to another [3]. The cars themselves were electrically-powered (which allowed them to be charged automatically upon arrival at the destination parking station) and the calculation of the usage time and the corresponding service charge were effectuated through very rudimentary yet functional technology [2]. Ultimately, a lack of political will/support led the "Witkar" cooperative to dissolve itself in 1986 [3].

Despite these setbacks, through the 10+ year operation of the Witkar cooperative Schimmelpennik had proven that this was possible, and he stood ready to contribute when the winds of change started to move towards more sustainable urban design principles, including the desire to reduce pollution and traffic congestion. An opportunity to do so arose in 1995 when two Danish entrepreneurs asked for his help to set up a bike-sharing system in Copenhagen, which culminated in the world's first large-scale bike sharing program [4]. Subsequently, in 2002 Schimmelpennik was enlisted by a French advertising company to set up additional such initiatives, first in Vienna and then Lyon [4]. These were both highly successful, leading to the launch of *Vélib* bike sharing system in Paris in 2007, whose success was unprecedented and catalyzed many cities around the world to implement their own bike sharing programs [4]. Indeed, the popularity of bike sharing systems increased massively in the 2010s. For example, in only three years from 2014 to 2017 the number of bike share programs worldwide increased 100% while the number of bikes available through such programs increased

[1]**Fun fact**: As of this past July, Google Maps has started incorporating bike share stations in its route planner!

[2]**Fun fact**: "Vélib" is a portmanteau of the French words vélo ("bicycle") and liberté ("freedom")!

2000% [5].[2]

1.2.3 Our Motivation

All of our group members were keen to do the *Bike Sharing and Rental Demand Estimation* project specifically. The principal factors behind our motivation for this project include a mutual interest in the sharing economy including the potential for viable business models therein, as well as a general interest in biking and the potential to increase the prevalence of bike-users through innovative approaches/models.

Additionally, from a data science perspective, the nature of the way bike sharing systems are set up (e.g. users check-in at a specific point, check-out at another, and their elapsed time-of-use is tracked) is quite conducive to generating a large amount of high-quality data. In turn, this data can be used to further enhance these systems' efficacy, as well as ameliorating the understanding of city movement patterns and informing city planning/design efforts accordingly.

2 Tech Stack

In his article titled "AI and HCI: Two Fields Divided by a Common Focus", the well-known researcher Jonathan Grudin considers the changing "seasons" of artificial intelligence (AI), from 1950 to 2009 (the article was published in 2010) [6]. Looking-forward, Grudin posits that a new "AI Summer" would arrive in the coming years [6, 7, p. 24].

To date, this prediction has indeed been borne out, and there have been many significant developments in the field of AI since Grudin wrote the aforementioned paper. For example, there has been a strong uptake in the number of businesses using machine learning technologies and artificial intelligence to ameliorate their businesses through the production of statistics that would help inform/enhance decision-making [7, p. 24]. In 2015 the decision-making process was mostly based on business reports, produced by "data scientists/data analysts", which leveraged the then state-of-the-art algorithms [8]. Just five years later, the decision-making processes now often require apps that: possess the capability to visualize findings, are interactive, and can be used throughout all levels of the company regardless of users' skill sets [8]. However, despite the significant advances in AI in recent years, the actual technological processes fulfilled by employees across a company can vary widely. Effectively integrating these processes requires strong collaboration between the different departments involved, such as the need for data scientists, data analysts, and developers to work together when creating an application [7, p. 22-23].

The rapid development of AI in recent years and the ensuing need for collaboration among different IT processes/functions has brought about certain challenges, such as reproducibility. Broadly, this is the need for "Programmers [who] need applications to run no matter where they are deployed" [8]. Imagine a Data Scientist developing an algorithm which he/she wants to implement in an application, which itself is managed by a Data Analyst. To do so, the Data Analyst must first get the desired algorithm up and running on his/her machine (and eventually in the cloud) [9]. One way to address the aforementioned problem regarding how to ensure effective collaboration between disparate functions/departments is the containerization of development environments [9]. In particular, an application developed within such a container is portable between different systems [9]. Moreover, these containers are easy to install and to use [9]. The idea of containerization, however, is only one of many tools that can be leveraged to ensure successful collaborative work. The containerization-related tools we employed during the course of this project are outlined in the **Reproducibility** section. Additionally, the tools we used to help our team collaborate effectively are detailed in the **Collaboration** section.

Chapter 2 concludes with the **Workflow** section which provides a description of the general workflow of our project, especially as it relates to the tools we used (and how said tools allowed us to work on the project in an effective manner).

2.1 Reproducibility

GitHub is a commonly used tool throughout the data science community. Some of the main reasons for its popularity include that it enables users to: rollback error-ridden code, merge the features they developed with code from other collaborators, and create an open code base so as to make the code be publicly available. Further details surrounding GitHub and how it works are provided in the **GitHub** subsection.

As mentioned, containers play a critical role in ensuring one's work can be readily reproduced, thereby helping to foster effective collaboration between team members. How containers work and how they can be used is described in the **Docker & Visual Studio Code** subsection.

Decision-makers now expect fully integrated applications that help enhance decision-making throughout all levels and functions of an organization [8]. Python is an example of one such a tool; it has the ability to "store, access, and manipulate data [out of the box]" and is thus widely used [10]. The various libraries it possesses for visualization, statistics, and standalone applications (e.g. flask) is one reason why our team made extensive use of Python during the course of this project. More information about Python and how we used it is provided in the **Python** subsection.

The last subsection of the **Reproducibilty** concerns SQLite. Commencing a Data Science/Analytics project typically necessitates the creation of quite a few files. These will often possess quite similar names such as *X_train_1* and *X_train_2*, making it easy for one to loose sight of what is actually in the files, which is of course quite pertinent for the main workflow. Over the long run this procedure creates unnecessary confusion, not only regarding naming but also through the sheer amount of files created. To address this challenge, our team made use of a SQL database. The advantages of such a storage mechanism are discussed in the **SQLite** subsection.

2.1.1 GitHub

The word GitHub, as one might guess, is a concatenation of two different words: Git and Hub. In order to understand this portmanteau, it is necessary to first understand how Git works. Broadly, Git is a version-control tool with two main functions, as described below.

The first use is at the individual level; each programmer can make use of Git when working on their own coding projects. Specifically, Git creates a snapshot of the written code and saves it in a .git-folder within the original folder. As a result, if for some reason a program breaks, one can always go back (to the code before the break), or even see the difference between the state of the working and development code. This makes it easier to identify the broken part of the more recently written code. When taking each snapshot (i.e. a "commit") the programmer himself/herself can write a message, wherein he/she clarifies the changes made. As needed, the programmer can access that log retroactively, so as to understand the state of the code at each commit and/or to which commit the programmer might want to consult.

Another use case, as Zhang et al. points out, is that nowadays there are various departments involved when developing applications [7, p. 25]. A prominent example of this is during the development of a business intelligence app. This process involves data analysts (who develop the different plots and their layout), data scientists (who develop the models to produce the later on visualized data), and the development engineers (who provide the fundamentals of the app). Though incomplete, this illustrates the fact that there are various people/functions working on different features of the same project. Git can be used in order to track the progress made by each coder/department, as well as when code merges occur and what these entail. Git provides the functionality to readily join different components of the code and notifies the user should there be a conflict therein. Once a code has been merged, all commit messages can be accessed, and the merge could be reverted if the programmer encounters problems.

Everyone who is working on a project has a copy (repository) from the main project on their local machine. He/she develops their features using the given source code and tracks his/her changes over time. Doing so immensely increases cooperation and collaboration when working together on a coding project.

The second part of the word GitHub is Hub. "Hub" refers to the free remote service, where all the code and the different branches can be stored. This remote repository enables all team members to access all components of the code as needed.

Once Git is installed on the local machine it can be accessed via the command line or the own graphical user interface, called GitBash. Through GitBash users can: create a git repository, commit, push to GitHub/GitLab, and pull code. This allows individuals who themselves may not have worked on the public project to pull (download) the code from GitHub and run it locally.

2.1.2 Docker & Visual Studio Code

In order to help facilitate a collaborative and efficient project execution, it is important that the infrastructure surrounding the project is designed in a way that allows every participant to readily reproduce the source code, without having to worry about how the software environment should look. One tool that is indispensable in this regard is Docker, as it "locks down" the software that is in active use and ensures each team member installs (and works with) the exact same set of software[3] [8]. The idea/concept behind what we refer to as a "container" is discussed in further detail below.

Big container ships bring goods from one

[3]Particularly regarding Python, it can be ensured that the same packages are installed within that environment using a requirements.txt.

port to another. These goods can include things like food, electronics, and furniture that require their own type of packaging; e.g. some have to be cooled while others may be fragile and hence require bubble wrap and foam. Despite the item-specific differences in packaging requirements, all of these goods can be transported on the same container ship. This is possible as a result of standardized shipping containers, which have the exact same dimensions and allow shippers to effectively and efficiently allocate goods between these containers. Due to their standardized size, these containers are easily loaded onto trains and/or trucks once the ship arrives in port, anywhere in the world. Docker converted this idea to the IT-World [11].

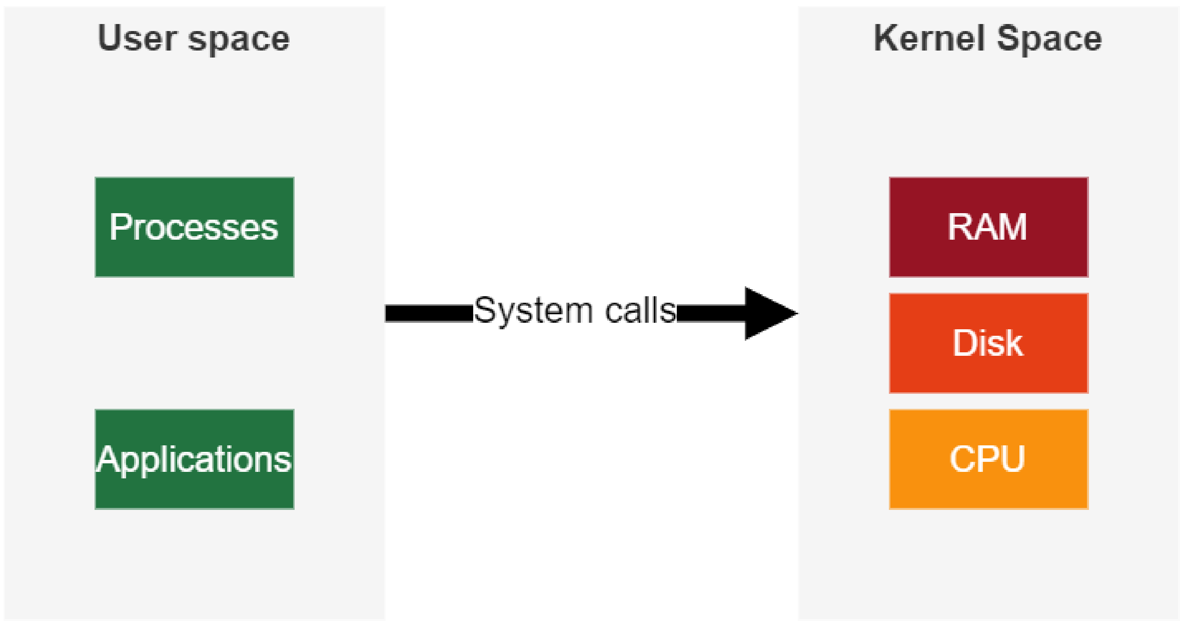


Figure 2: User and kernel space in a Docker container [12]

The idea of containers makes use of two core Linux concepts - the user space and the kernel space [12]. As illustrated by Figure 2, the user space contains all processes and applications initiated by the user (e.g. creating a file) while the kernel space receives these processes and interacts with physical devices such as the disk, RAM, and CPU [12].

Taking heed of this concept, Docker defined a container as a process that is able to access the kernel space. In that way, several containers with different layers can run (e.g. Ubuntu, CentOS) simultaneously on a Windows host. Those processes use a *Dockerfile* or a *yml* file, where the different layers of that container are defined. The kernel space defines how much RAM, disk and CPU power each process gets, though the user also has the ability to customize these values. The usage of these concepts also makes it possible to run these containers on almost all host systems (e.g. MacOS, Windows, Linux).

In 2019 Microsoft published an extension for their program Visual Studio Code which was called *Containers - Remote* [13]. This extension for their code manager allowed users to create a *devcontainer.json* and through this, created a container environment within Visual Studio Code (VS Code). As a result, users in this *devcontainer.json* environment can specify: which docker file should be used to create the container, which extensions should be installed in VS Code, and the specific ports available to the local machine while the container was setup. Overall, this extension has significantly simplified working with containers in a shared environment. How exactly it simplified the workflow is further discussed in the **Workflow** subsection.

2.1.3 Python

Python is a general-purpose programming language that is especially popular within the fields of Machine Learning and Artifical Intelligence. As Python is an open source project from Guido van Rossum, the online community has created various libraries which help while programming in Python (the most well-known such libraries include pandas, sckit-learn and numpy). Nowadays, Python is the go-to language for predictive statistics, so it should come as no surprise that we make extensive use of Python in our efforts to accurately predict bike share rentals.

2.1.4 SQLite

Typically, **S**tructured **Q**uery **L**anguage (SQL) is used to query **R**elational **D**ata**B**ase **M**anagement **S**ystems (RDBMS) such as Oracle. In other words, SQL is a tool used to make queries, and is not itself a database. That said in 2000 SQLite was published; this new library was a standalone SQL database, which could be readily created and queried [14]. When used, SQLite creates a file on the disk, which contains all the information regarding the tables and their schema, just as a normal database would. According to the information on the SQLite website, SQLite is now one of the most used databases in the world [15].

This general library is also available in Python, where it is called sqlite3. The project *Bikerus* made use of this library. By doing so we were able to easily create and connect to the database. For example, the single line of code shown below creates the *BikeRental* database, assuming it does not already exist. If it does it will create a connection object, and this can be used to query the database.

```python
import sqlite3
connection = create_connection
    ↪ ("./database/BikeRental.
    ↪ db")
```

Code Excerpt 1: Creating a connection to SQLite database

Now, using this connection object, one can query the database and save or receive tables using the *pandas* package.

```python
import pandas as pd
# read data from sqlite
    ↪ database
df = pd.read_sql_query('''
    ↪ SELECT * FROM hours''',
    ↪ connection)

# post data to sqlite database
df.to_sql("hours_complete",
    ↪ connection, if_exists="
    ↪ replace", index=False)
```

Code Excerpt 2: Read and post to SQLite database

Using the *sqlite3* library significantly reduced the volume of files we had to interact with, because it stored all the data as tables within the database. Moreover, it made every table accessible and human readable, rather than just checking whether or not a .csv file has the appropriate columns and if they are filled correctly.

2.2 Collaboration

Up to now we have outlined the tools we leveraged to help ensure our work was reproducible. In addition to the importance of reproducibility in the context of this kind of group work, effective project execution also required us to establish an organized work stream, where each team member's individual tasks are clearly delineated. Moreover, we needed some way to properly document our findings. For the first need (regarding the development of an organized/efficient work stream), we required a tool able to incorporate our ideas in terms of tasks that needed to be completed, to which we could then assign deadlines and allocate to the appropriate team member. To do so, we made use of a publicly-available project management tool called ”Asana”. Further details concerning how exactly we used this tool during the course of the Bikerus project are provided in the Asana subsection.

As mentioned, we also needed some mechanism through which we could produce editable academic text, allowing each team member to see what their fellow collaborators have written. A solution that does so is Overleaf, which is a Latex based editor. The advantages it brings and how it was used within this project are explained in the LaTeX subsection.

2.2.1 Asana

The first step of each group project is the brainstorming; gathering ideas, considering how to approach the topic, thinking about the different tasks involved, considering the end goal, and determining near-term objectives and ”deadlines”. To do so, we made use of Asana. Asana is a project management tool, which allows users to create different tasks with deadlines and assign these tasks to the applicable team member(s). Asana allows each team member to access this task board and determine which tasks are currently in process, which have been completed, and/or those tasks that have yet to be assigned.

Within Asana, users can create different boards, with each board representing a different project or a distinct step within a project. Furthermore, each task can contain several subtasks, which again can be assigned to different team members. Thus, this tool provides a structure that is especially helpful in the early stages of group work, and we used it heavily during the *Bikerus* project.

2.2.2 LaTeX

The last application of the *Bikerus* Tech Stack is Overleaf. This application offers a platform for working collaboratively on scientific documents [16]. Since it is a cloud platform, all changes are available for all members in real time. This simplifies working together and helps avoid circumstances wherein two team members accidentally repeat each other's work. Moreover, Overleaf offers a wide variety of input settings, from including graphics, to writing computer programming code and complex mathematical formulas. For more information on the various applications of Overleaf please refer to the citation provided above.

2.3 Workflow

In the preceding sections we have outlined the tools used by the *Bikerus* team in order to ensure (1) the reproducibility of our work and (2) effective team collaboration. In this section we outline the approximate chronology of when we used each tools during the course of the project.

As mentioned, the first step in this project was to get together and brainstorm our ideas regarding how to approach the overarching task of predicting bike share rentals. All team members had the opportunity to put forward their ideas, and we determined as a group which ideas/tasks should be inputted to Asana. For this project, we created three different boards: Data Science, Presentation, and Paper. We created these three boards to reflect the fact that the requisite tasks would vary as we progressed through the different stages of the project, as can be seen in the below Figure 3.

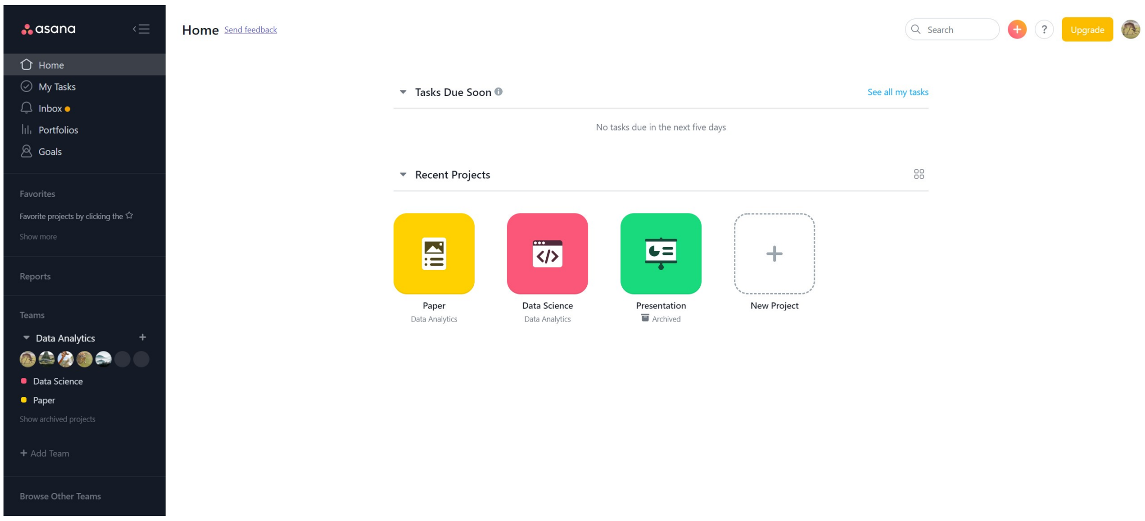


Figure 3: Asana dashboard

We then created a new GitHub repository and in it, a devcontainer.json and a dockerfile; doing so helped ensure each team member was working under the same software conditions. Specifically, each team member cloned this git repository: *Bikerus*. Thereafter, each member created their own branches, thereby ensuring that nobody worked on the master branch, as the master branch should always be the functional and tested branch. Once the repository had been cloned, the folder had to be opened in VS Code. VS Code automatically detected that a container could be created, using the given devcontainer.json file. As a result the only thing the user had to do was to click "*open in repository*". This either built the container with the given requirements or started the container, depending on whether this was the first time the team member had made use of this functionality. Whenever a team member needed to use a specific Python package, he could update the *requirements.txt* file, in which all package dependencies are written down, and rebuild the container. During the process of rebuilding the container, all packages within this text file are installed in the container. Once a task had been completed, the branch was committed and pushed back to GitHub, and the applicable Asana task workflow board would be updated accordingly. On Github the pushed branch would be merged with the master branch; this was done by a specific team member who had been assigned to this task. This process was repeated as often as needed and until all tasks outlined in Asana had been completed.

Overleaf was used once we got to the drafting stage of this project. Overleaf allowed each team member to view in real-time the work of the other members, and make suggestions/edit others' work as appropriate. Moreover, once it had been properly set-up, Overleaf took care of code formatting, image insertion and transformation, and type setting. Indeed, the report you are reading now was entirely developed within Overleaf.

3 Data Preprocessing

3.1 Overview of the Dataset

We were dealing with multivariate time series data with different variables one would expect to impact the demand for bike-share rentals, largely environmental and seasonal factors such as temperature and windspeed.

The time period represented by the data was from January 1, 2011 to December 31, 2012, broken down by hour (each row represents one hour). Capital Bikeshare was launched in September 2010, so our dataset provided information pertaining to the relatively early stages of Capital Bikeshare's operation.

Initially, we did not think there were any missing values (henceforth referred to as "NAs"), because the data contained $17,379$ rows, none of which included any NAs. However, when we calculated the total number of rows that our dataset should include, it became clear there were in fact some NAs. Specifically, our dataset covered the 365 days in 2011 and the 366 days in 2012 (which was a leap year), meaning the total number of rows that should be included therein was $(365 + 366) * 24hrs$ per day, which equals $17,544$. This meant our dataset contained 165 missing rows ($17,544$ required rows less the $17,379$ rows that were present), so our first step when processing the data was to impute values for these 165 missing rows.

3.2 Imputing NAs

We started by creating a new "Date-time" index that showed the date (including the day of the week) and the time of day. Through this function, we created the 165 missing rows that needed to be filled, populating the missing date, hour, month, and weekday values. For the remaining NAs, we employed the Pandas package to forward fill certain data elements and interpolate others. Specifically, we used forward fill when inputting data elements that would not change from one row (hour) to another, e.g. year, season, holiday. A visualization of the forward fill function is provided in Figure 4. On the other hand, if it was likely that the data would change on an hourly basis (e.g. temperature, humidity, windspeed) then we would use interpolation to impute the NAs.

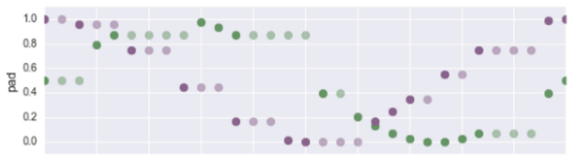


Figure 4: Visualization of the forward fill function

In the Pandas' interpolation function, it is possible to set a limit for consecutive rows with NAs. For example, if the limit is set to two and we have three consecutive rows with NAs for one or more columns, the interpolation method will only fill the first two rows. One can also set this parameter when using Pandas' forward fill (ffill) function. However, unlike the interpolation method which is very good in predicting missing values based on the observed values before and after NAs, the forward fill function is not sufficiently 'intelligent' to make predictions for missing values. Instead, this function just copies the previous row's data and drops said data into the empty row [17]. In Figure 5 you can see how the interpolation function imputes missing values aligned with the existing data points.

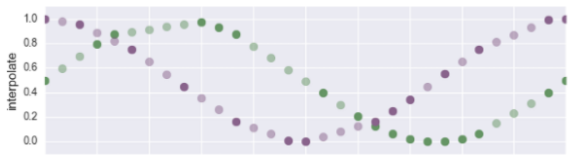


Figure 5: Visualization of the interpolation function

After imputing our dataset, we needed to carefully review the completed dataset to ensure the time series is correct as such datasets have very sensitive behavior (with respect to autocorrelation). To do so we used the Pandas Profiling package, which can automize the creation of plots and basic statistics. Through our understanding of the multivariate time series dataset, we were able to determine whether each month has the correct amount of data by assessing if the hour data is spread evenly and if we have the correct season value. Firstly, we can simply divide the number of rows for each of the two years (year = 0 or year = 1) by 24. The results should be 365 and 366 respectively. By doing so we deduced that the season data spread is not even. Therefore, we had to check when the season information changes. We figured out that the season shifts usually around the 20th/21st of March, June, September, and December. This is why the season information isn't distributed evenly, yet nonetheless our forward fill function completed the time series correctly (verified manually).

3.3 Data Visualization

In both academia and business, the importance played by data visualization in the context of exploratory data analysis and result presentation is well-understood [18]. Accordingly, the following subsection of this paper provides several visualizations regarding our dataset, and considers certain insights gleaned from these visualizations. We used exploratory data analysis to gain a preliminary understanding of some of the main characteristics of the dataset, which along with some degree of domain knowledge helps to inform potential actions such as dropping or engineering specific features. The point of this section is not to provide an exhaustive graphical analysis, but rather to serve as a means through which readers can readily understand key elements of our dataset, in conjunction with the Overview of the Dataset section.

The data visualization approach we followed is based on two principal components: (1) The target variable which was always kept on the graphs' y-axis, and (2) a time-based feature that was kept on the x-axis. By increasing the level of time granularity graph by graph ("zooming in" on the time aspect of the data via seasons → months → hours), we essayed to gain further insights into the relationship between the target variable and the respective time variable. For all plots, we used Seaborn; a popular Python data visualization library based on matplotlib.[4]

To begin, we looked at how the target variable (average user count) behaves across the seasons: Spring (1), Summer (2), Autumn (3), Winter (4).

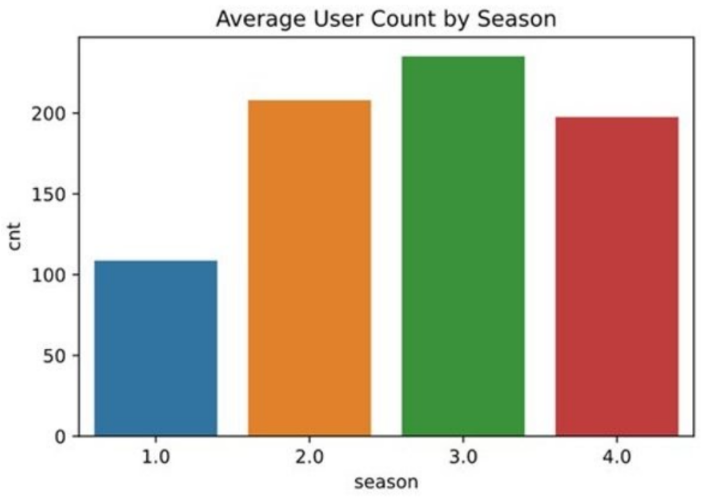


Figure 6: Average user count by season

As demonstrated by Figure 6, bike rental demand is higher in the middle of the year than it is at its start and end. Further details regarding this general observation are provided by Figure 7 and the increased granularity it provides by looking at months rather than seasons. Furthermore, these two graphs also suggest a relationship between the target variable and the weather and temperature. Washington, D.C. exhibits subtropical climate [19] that is accompanied by more "biker-friendly" weather conditions in the middle of the year, as is typical for regions on the Northern Hemisphere. A separate analysis of the weather and temperature aspect is beyond the scope of this time-based graphical analysis but could be insightful for future exploration.

Next, we zoomed in even further on the time aspect by looking at the average user count by the time of day. Doing so clearly identifies strong variations in bike rental usage based on the time of day.

[4]To increase readability we refrained from displaying the code for the sanity checks and visualization.

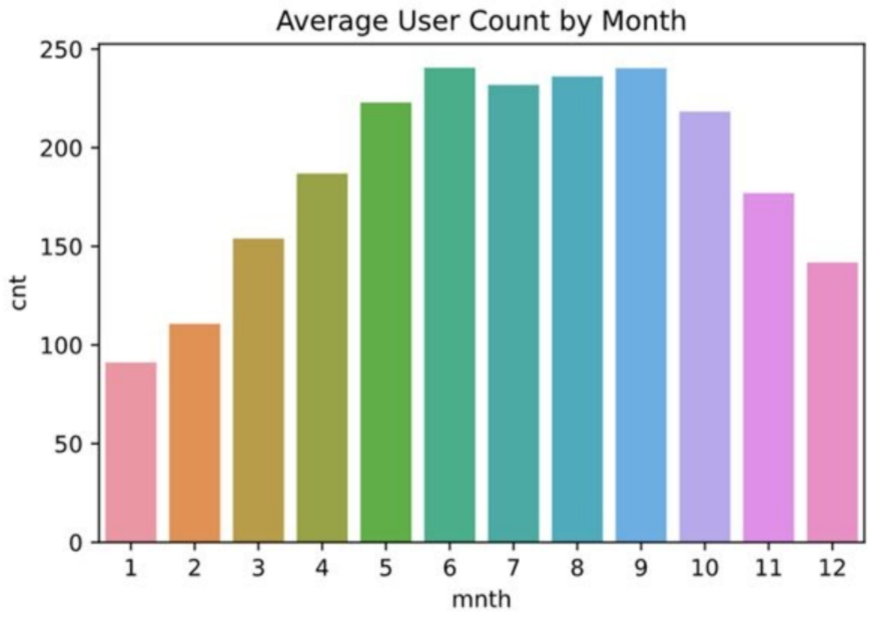


Figure 7: Average user count by month

There are significant spikes in bike rentals around the start and end of the typical working day (what we refer to as the ”rush hour effect”) with a lower plateau of bike rental demand in between those spikes and highly depressed demand after approximately 22:00 until about 7:00.

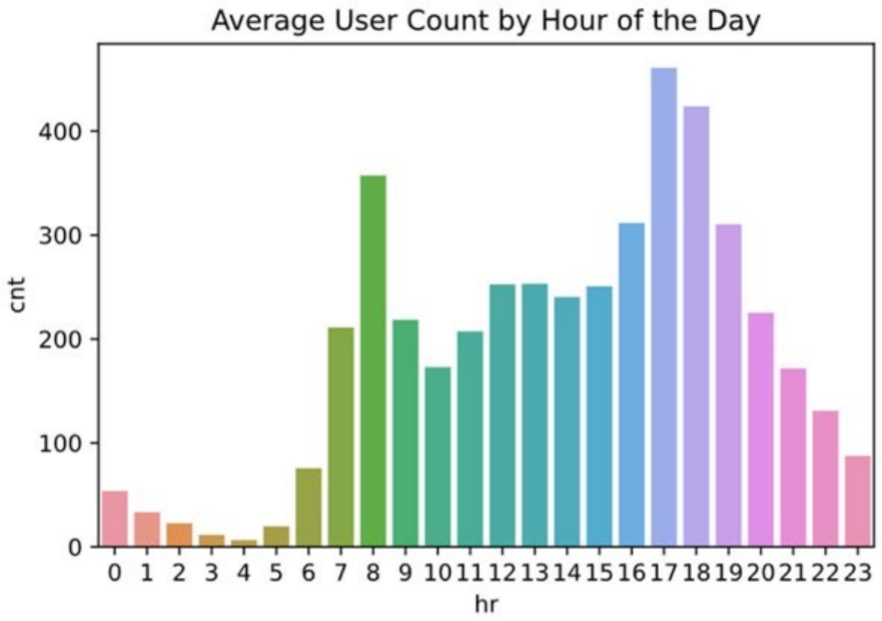


Figure 8: Average user count by time of day

The rush hour effect can be seen even more clearly when we separate working days from weekends and holidays. On weekends and holidays, bike rental demand resembles a bell curve centered around midday whereas on working days the aforementioned rush hour effect around 8:00 and 17:00-18:00 becomes even starker. We will keep this specific observation in mind when it comes to feature engineering.

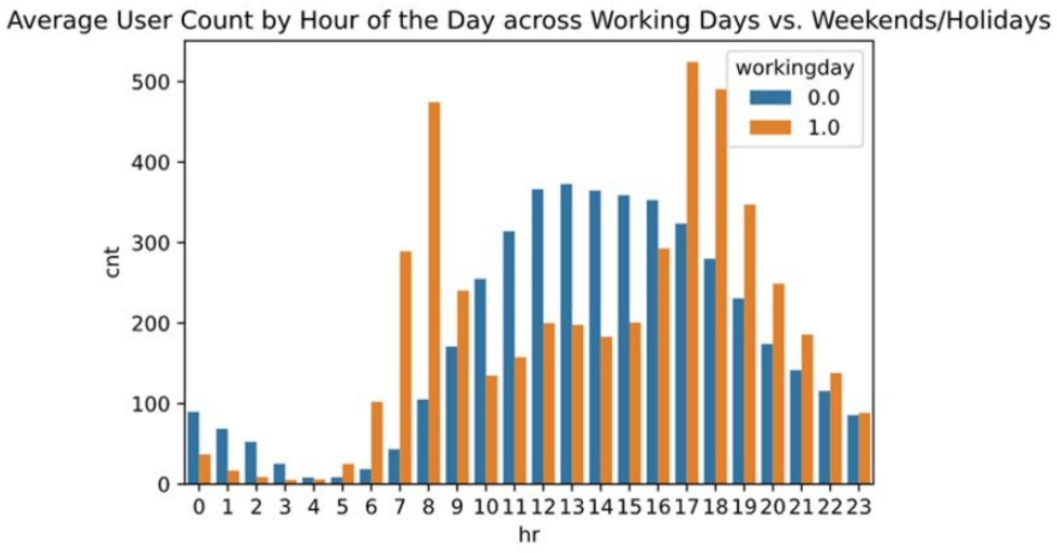


Figure 9: Average user count by time of day on working days vs. weekends & holidays

Another important aspect of any given dataset is the correlation (or lack thereof) of its variables. This can be clearly visualized using the correlation matrix as a heatmap which shows the pairwise Pearson correlation of the dataset's continuous variables. Such a heatmap is displayed in Figure 10.

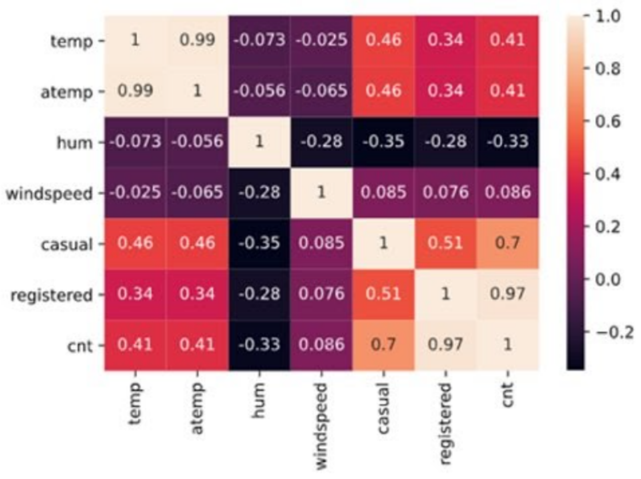


Figure 10: Correlation matrix of continuous variables

The almost perfect linear correlation (0.99) between the temperature (*temp*) and the feeling temperature (*atemp*) immediately stands out. This observation will be pertinent in the following section: Feature Selection & Engineering. Another notably high correlation can be seen between the different user counts: *casual*, *registered* and *cnt*. Again, this suggests further analysis into these relationships is warranted, and we will do so in the following section.

To sum up, this brief graphical exploration of the data not only provided us with some insights on the relationships between some of the dataset's components (time and bike rental demand) but also highlighted the rush hour effect and indicated striking correlations that should be analyzed during further preprocessing of the dataset.

3.4 Feature Selection & Engineering

3.4.1 Dropping

After imputing the NAs we had to dig into our data and decide which features were really necessary and useful. We started by investigating the difference between the *temp* and *atemp* columns since both provide information regarding the temperature. The *temp* column represents the normalized temperature in Celsius and the *atemp* column represents the normalized "feel" temperature (i.e. what the temperature actually feels like after having accounted for such factors as wind speed and humidity) in Celsius. Both of these features were highly correlated so we decided to drop the *temp* feature. We did so because we think how the temperature feels would have a greater impact on whether people decide to rent bikes than does the unadjusted temperature reading.

Additionally, our target value (hourly bike rentals) was split into two other features: casual and registered users. If we were so inclined, we could have developed machine learning algorithms for each of these two features. Instead, we decided to focus on a single aggregated variable that took the sum of the casual and registered users, and we dropped the individual casual and registered columns. We also dropped the *dteday* feature, because we wanted to prohibit the algorithm from drawing conclusions for anything other than the exact date.

Last but not least we discovered that throughout the entire two year period there were only three rows (hours) where the weather had been classified as category four (severe/very bad). Accordingly, we treated those three hours as outliers and deleted the corresponding rows from the dataset.

3.4.2 Engineering

As mentioned, we were working with a multivariate time series dataset. When visualizing certain elements of this dataset, we noticed there were similar patterns for some variables depending on whether it was a holiday or a working day. For example, during working days we identified "rush-hours" from 7:00-8:00 and 16:00-18:00. During these time frames there were clear increases in the demand for bike rentals, ostensibly because of the commute to and from the workplace. To support our algorithms to identify these time periods, we added a column with boolean data type with true indicating that the current row represents a rush hour. Doing so increased our models' R^2 and Pseudo-R^2.

3.4.3 Normalization

If features exhibit different value ranges, it might be a good idea to normalize their values to guarantee optimal incorporation within machine learning models [20]. The goal pursued by normalization is to align the features' scales (between 0 and 1) without changing the relative differences between the values. In our case, the assignment sheet stated that the weather features were given in normalized form, but upon closer inspection we saw that the minimum value for temperature was 0.02 and the maximum humidity value was 0.85. Although these are no dramatic scale differences, it would be imprecise to decide against applying correct normalizations to all continuous variables. The normalization was carried out by using Scikit-learn's MinMaxScaler [21] which uses the following formula:

$$x_{scaled} = (x - x_{min})/(x_{max} - x_{min})$$

It is important to store the original minimum and maximum value of the target variable in order to revert the regression models' estimations back to the original scale and make the results interpretable by rearranging the formula. Normalization was also done for the specific data preprocessing for non-tree approaches (refer to chapter 3.5).

3.5 Preparing Data for Non-Tree Approaches

There are several ways to distinguish different groups of machine learning models to use for classification or regression problems. One important distinction with regards to data preparation is whether the selection of models include both tree-based and non-tree-based approaches. The reason for that is that they require differently preprocessed data. While tree-based models do not require specifically encoded categorical variables since they can handle different values that stand for a specific categorical feature (e.g., spring: 1, summer: 2, autumn: 3, winter: 4), non-tree approaches use the data instances' distances and numeric values to incorporate them into the model. Since we wanted to use the non-tree approaches of neural networks and support vector regression, we had to examine the categorical (and especially cyclical) variables that need specific treatment in order to be used by these models in the best fashion. Technically we created separate preprocessing steps for non-tree approaches within the preprocessing pipeline.

Based on the preprocessing that was already done, we looked at all features and identified the optimal procedure of encoding for non-tree approaches. Three popular ways of encoding categorical variables are integer encoding, one hot encoding, and entity embedding [22]. Integer encoding maps every category label to an integer. It can be used for non-tree approaches when it's paired with the normalization of the integer scale. In our dataset, the season feature is an example of non-normalized integer encoding. One hot encoding (OHE), represents each category label as a binary value (either 0 or 1). This binary approach solves the problem of integer encoding that category labels with higher values might be interpreted as more important or "better". Entity embeddings are distributed representations of categories that are suitable for categories with high cardinalities. Furthermore, we had to identify cyclical data features and give them special treatment in order to address the problem of inconsistent distances within integer encoded cyclical features [23]. Our approach of applying sine and cosine transformations (sin/cos) in conjunction with normalization will be looked at more closely below.

Treatment of Features	
Feature	**Treatment**
datetime	**drop**
season	**sin/cos**
year	already OHE
month	**sin/cos**
hour	**sin/cos**
holiday	already OHE
weekday	**sin/cos**
workingday	already OHE
weather sit	**OHE**
rush hour	already OHE

As we can see from this overview, a lot of features were already in binary (OHE) form and thus did not need further treatment since non-tree approaches are able to handle them. As well as for the tree-approaches the datetime feature had to be dropped since the models we will use are not compatible with the datetime format. More importantly, there are two transformations that we had to do: OHE for the weather situation variable and sin/cos for cyclical features.

Based on the previous preprocessing steps, the weather situation was still integer encoded, so it needed some further preprocessing to be best incorporated in non-tree models. Since the weather situation is not a pure cyclical feature, such as months for example (see below), we decided to use simple OHE using pandas' *get_dummies* while avoiding the dummy variable trap by dropping one of the resulting columns [24].

```
# create dummy variables for categorical weathersit while
  ↪ avoiding dummy variable trap
df_dummies = pd.get_dummies(
    df["weathersit"], drop_first=True, prefix="weathersit")
df = df.drop(["weathersit"], axis=1)
df = df.join(df_dummies)
```

Code Excerpt 3: One hot encoding of weather situation

Since our dataset included several cyclical features we had to find a preprocessing strategy in order to ensure best possible use of the data even for non-tree approaches. The problem of non-treated cyclical features in non-tree models can be shown by a simple example. Without any further preprocessing the hour features ranges from 0 to 23, representing each hour of the day. Based on human knowledge and experience with time, we know that the hour 23 and the hour 0 are close to each other which might let us derive that bike rental demand in these hours are likely to be similar. While this might be easy to interpret as a human being, non-tree machine learning models only know that these values are far away from each other (maximum distance within the range). This does not change if the data gets normalized because only the scale is changed. Therefore, we need a transformation that assigns an equal distance between adjacent values. This can be done by applying sine and cosine transformations to each cyclical feature resulting in two new columns (sine & cosine) for each original cyclical column [23].

```
# apply cos & sin transformation for cyclical features
cycl_var = ["season", "mnth", "hr", "weekday"]
mm_scaler = preprocessing.MinMaxScaler()
for i in cycl_var:
    df[i] = df[i].astype("int32")
    df[f"{i}_sin"] = np.sin(2 * np.pi * df[i]/df[i].nunique())
    df[f"{i}_cos"] = np.cos(2 * np.pi * df[i]/df[i].nunique())
    sin_cos = [f"{i}_sin", f"{i}_cos"]
    df[sin_cos] = mm_scaler.fit_transform(df[sin_cos])
    df = df.drop(i, axis=1)
```

Code Excerpt 4: Sine cosine transformation of cyclical features

If we take a look at the example variables of hour and season after applying cosine and sine transformations and normalization, we can clearly see how the adjacent values remain at a constant distance to each other by being mapped on a circle. For example, the aforementioned problem regarding hours 0 and 23 was solved by doing so. The same holds for all other cyclic variables, as can be seen in Figure 11.

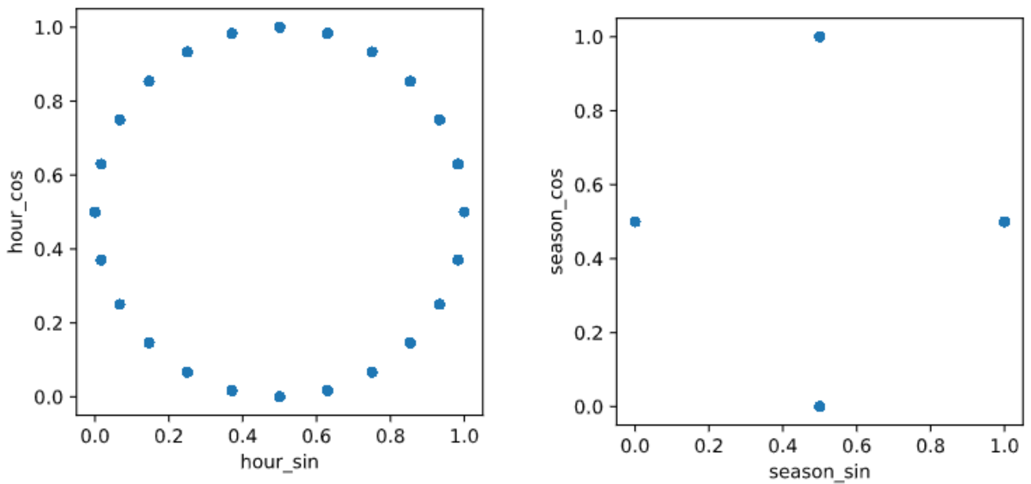


Figure 11: Sin/Cos transformed & normalized hour & season feature

3.6 Data Partitioning

The first step when partitioning the data was to separate the dependent target feature *cnt* from the rest of the data. Accordingly, we separated the target feature *cnt* as Y from the rest of the data, as X. The second step was to divide X and Y into distinct training and test samples. To do so, we followed two approaches: (1) Random Train-Test-Split and (2) Time-Based Train-Test-Split.

3.6.1 Random Train-Test-Split

In our first approach, we randomly partitioned the entire dataset into training and testing sets using the *train_test_split* method implemented in scikit-learn as shown in code excerpt 5. *Train_test_split* assumes that there is no relationship between the observations, i.e. that each observation is independent [25]. Following a review of the available literature, we elected to use a 20% test sample size.[5] Additionally, we set the parameter *random_state* to 0, thereby ensuring reproducibility when calling the function multiple times (i.e. the split will be identical when calling the function a second or third time) [29].

```
def train_test_split_rs(data, train_size):
    X = data.drop(['cnt'], axis=1)
    Y = data['cnt']

    X_train_rs, X_test_rs, Y_train_rs, Y_test_rs =
  ↪ train_test_split(X, Y, train_size = train_size,
  ↪ random_state=0)

    return X_train_rs, Y_train_rs, X_test_rs, Y_test_rs
```

Code Excerpt 5: Random train-test-split

3.6.2 Time-Based Train-Test-Split

For the second approach employed to partition the data, we determined that the training sample should constitute the first 80% of our data, with the remaining 20% of our data therefore encompassing the test sample. The dataset is a sequence of data points indexed in time order, i.e. the dataset consists of time series data. As previously stated, the *train_test_split* approach partitions the data randomly. As a result, data from December 2012 (training set) may for example be included in a model trained to make predictions about periods prior to December 2012 (test set). This does not accurately resemble the situation in a production environment, where, once a model has been adequately trained, it makes (future) predictions based on past data.

To remedy this, we partitioned the data based on the time variable. By doing so we ensured that we did not use information about the future to make inferences regarding past events. Again, we set the parameter *random_state* to equal 0 so as to allow the function to be accurately reproduced.

[5]This review included the following resources: [26], [27], and [28].

```python
def train_test_split_ts(data, train_size):
    X = data.drop(['cnt'], axis=1)
    Y = data['cnt']
    index = round(len(X) * train_size)

    X_train = X.iloc[:index]
    Y_train = Y.iloc[:index]
    X_test = X.iloc[index:]
    Y_test = Y.iloc[index:]

    return X_train, Y_train, X_test, Y_test
```

Code Excerpt 6: Time-based train-test-split

3.6.3 Cross Validation

To better evaluate the performance and generalizability of our models, we further split the training data into smaller subsets (called "folds") to use them for cross validation. This allowed us to make better use of our data, as each observation in our training set is used for training and testing [30]. Additionally, doing so enabled us to tune the hyperparameters for each model, e.g. the number of trees in the *RandomForestRegressor* [30].

Using cross validation, we applied multiple hyperparameter combinations to each model on the training set and estimated the models' performance, making adjustments to the hyperparameters as needed. This allowed us to use the original test just once for a final analysis of the generalizability and of the performance of our models. Otherwise, had we not incorporated cross validation, we would not have been able to adequately evaluate our models and adapt the hyperparameters accordingly; there would have been a greater risk that our models would not perform well when faced with new data.

We followed two approaches when conducting cross validation: GridSearchCV and TimeSeriesSplit. Based on some external research, we elected to use five folds for cross validation for both GridSearchCV and TimeSeriesSplit [31, 32, p. 242].

GridSearchCV

Scikit-learn's GridSearchCV uses a (stratified) KFoldcross-validation splitting strategy to find the best hyperparameters [33]. This strategy randomly divides the training data into k folds of equal size. The models are fitted using k–*1* folds, while the remaining fold is used for testing. We used GridSearchCV for both of the initially created test-sets, i.e. the test set based on a random train-test-split and the train-test-split based on time.

Refer to chapter 4 for the different hyperparameter combinations used throughout cross validation for each model.

TimeSeriesSplit

Additionally, we elected to make use of the *TimeSeriesSplit* function from sklearn in order to account for the time series character of our dataset during cross validation. *TimeSeriesSplit* is a variation of KFold; it treats successive training sets as supersets of the preceding training sets [34].

We incorporated our approach by means of an additional function *get_sample_for_cv*, which created the train and test sets used for cross validation as documented in code excerpts 7 and 8. Creating a separate function increased the efficiency of our pipeline. Instead of copying the same function multiple times, it could just be called when training the different models.

The function included six parameters, of which only the last two (*X_test*, *vis*) are necessary for the creation of the horizontal bar diagram to visualize the testing iterations (refer to Figure 12). *n_splits* determines the number of splits used for cross validation (which must be greater than 1). *fold* specifies the current fold of the train- and test-set used for cross validation; it must be greater than 0 and not greater than the number of splits. *X_train* and *Y_train* represent the dataframes used to train the model.

First, the function determines the indices for each train and test fold and stores them in a list called *list_tscv*:

```
def get_sample_for_cv(n_splits, fold, X_train, Y_train, X_test
    ↪ =False, vis=False):

    # Creation of train and test sets for cross validation
    tscv = TimeSeriesSplit(n_splits=n_splits)
    list_tscv = []
    for train, test in tscv.split(X_train):
        list_tscv.append([test[0], test[-1]])
```

Code Excerpt 7: Folds for cross validation using *TimeSeriesSplit* - part 1

list_tscv consists of five sublists, each indicating the first and last element of the corresponding test-set: [[2340, 4678], [4679, 7017], [7018, 9356], [9357, 11695], [11696, 14034]]. Based on these indices, the function *get_sample_for_cv* creates and returns the four datasets required for training and cross validation: *X_train_current* and *Y_train_current* as well as *X_test_cv_current* and *Y_test_cv_current*.

The following example illustrates the functionality of *get_sample_for_cv*: For the first fold of a five-fold cross validation, i.e. *n_splits* = 5 and *fold* = 1, the function would return training sets consisting of the first 2,339 observations and testing sets consisting of observations 2,340 to 4,677.[6]

```
    if n_splits == fold:
        X_train_current = X_train.iloc[:list_tscv[fold-1][0]]
        Y_train_current = Y_train.iloc[:list_tscv[fold-1][0]]
        # +1 to include the last element X_train
        X_test_cv_current = X_train.iloc[list_tscv[fold-1]
                                         [0]:list_tscv[fold
    ↪ -1][1]+1]
```

[6]To increase readability we refrained from displaying the code for the sanity checks and visualization.

```
        # +1 to include the last element of Y_train
        Y_test_cv_current = Y_train.iloc[list_tscv[fold-1]
                                          [0]:list_tscv[fold
↪  -1][1]+1]
    else:
        X_train_current = X_train.iloc[:list_tscv[fold-1][0]]
        Y_train_current = Y_train.iloc[:list_tscv[fold-1][0]]
        X_test_cv_current = X_train.iloc[list_tscv[fold-1]
                                          [0]:list_tscv[fold
↪  -1][1]]
        Y_test_cv_current = Y_train.iloc[list_tscv[fold-1]
                                          [0]:list_tscv[fold
↪  -1][1]]

    X_train_current.to_sql("X_train_current", connection,
                           if_exists="replace", index=False)
    Y_train_current.to_sql("Y_train_current", connection,
                           if_exists="replace", index=False)
    X_test_cv_current.to_sql("X_test_current", connection,
                             if_exists="replace", index=False)
    Y_test_cv_current.to_sql("X_test_current", connection,
                             if_exists="replace", index=False)

    return X_train_current, Y_train_current, X_test_cv_current
↪  , Y_test_cv_current
```

Code Excerpt 8: Folds for cross validation using *TimeSeriesSplit* - part 2

Figure 12 below provides a sense of the breakdown between the training and testing datasets we used for cross validation using *TimeSeriesSplit*. The bars numbered one through five represent individual time series splits used for cross validation, whereas the *Final* bar embodies the partitioning for the final testing of the model. We trained the model a final time on the full training set and estimated the model's generalizability with the testing set. The relatively high proportion of test set data in the *Final* bar is indicative of the aforementioned parameter we set that 80% of the data be training data and 20% be used for testing (refer to chapter 3.6.2).

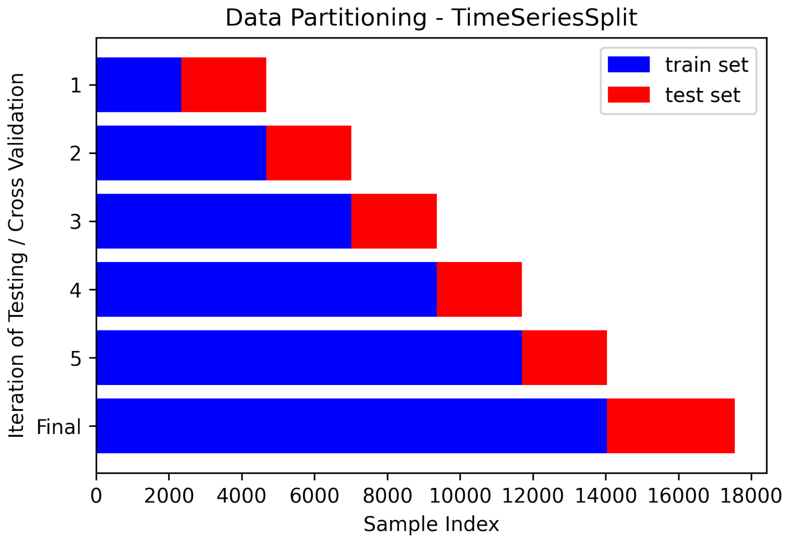


Figure 12: Data partitioning - *TimeSeriesSplit*

We implemented the cross validation based on *TimeSeriesSplit* using cascaded for-loops. Below we illustrate our approach for the *RandomForestRegressor*. That said, the explanations also hold for the cross validation used for the other models, the only difference being that other hyperparameters were applied.

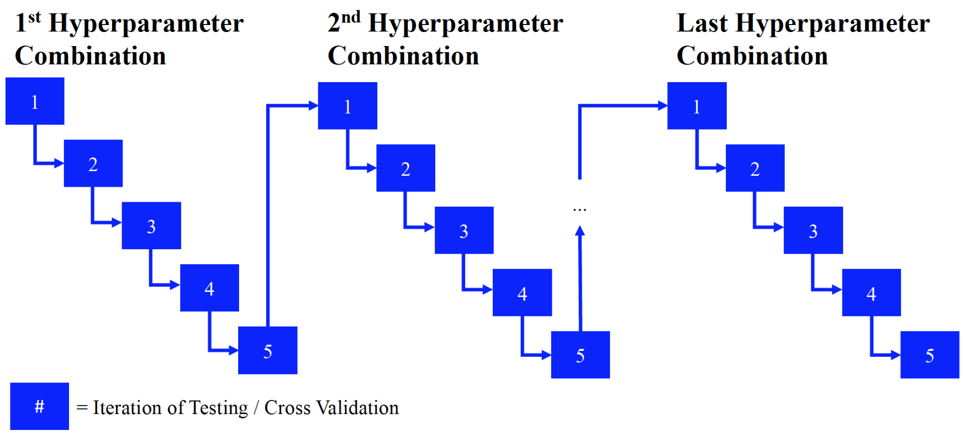
After initializing an empty dataframe which will store the R^2 and Pseudo-R^2 for each fold and hyperparameter combination, the values for each hyperparameter must be specified. The different hyperparameter combinations are applied to the model through the aforementioned cascaded for-loops.

```
df_parameters = pd.DataFrame()
folds = list(range(1, 6))
max_depth =  [8, 9, 10, 11, 12]
n_estimators = [100, 120, 140]
max_leaf_nodes = [60, 70, 80]
max_samples = [0.1, 0.2, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99]
for depth in list(range(len(max_depth))):
    for number_trees in list(range(len(n_estimators))):
        for node in list(range(len(max_leaf_nodes))):
            for sample in list(range(len(max_samples))):
                for fold in list(range(len(folds))):

                    X_train_cv, Y_train_cv, X_test_cv,
   ↪ Y_test_cv = get_sample_for_cv(folds[-1], folds[fold],
   ↪ X_train, Y_train)
```

Code Excerpt 9: Cross validation using *TimeSeriesSplit* - part 1

The last for-loop always needs to be the one indicating the respective fold. It establishes the order of the iterations and calculations, thereby ensuring that a given hyperparameter combination is applied to all different folds before another hyperparameter combination is chosen as is illustrated in Figure 13. This design is important for determining the order of the data rows in the dataframe *df_parameters* and the subsequent calculation of the mean of the Pseudo-R^2 used to identify the best hyperparameter combination.



1st Hyperparameter Combination
2nd Hyperparameter Combination
Last Hyperparameter Combination
1
2
3
4
5
...
# = Iteration of Testing / Cross Validation

Figure 13: Sequence of the application of the hyperparameter combinations

We decided to use R^2 and Pseudo-R^2 instead of the mean squared error (MSE) as a performance metric during cross validation using *TimeSeriesSplit*. R^2 returns a value between 0 to 1 indicating the percentage of variance in the dependent variable that can be explained by the model, whereas MSE is an absolute measure of the quality of fit [35]. Therefore, we chose R^2 and Pseudo-R^2 due to their enhanced interpretability.

Those metrics were calculated for each fold and hyperparameter combination. After calculating the mean value and the residual sum of squares of the target variable as a benchmark for the specific fold, the model is initialized with the respective values for the different hyperparameters. The model is trained, predictions are made and the R^2 and Pseudo-R^2 are calculated and appended to the initially created dataframe *df_parameters*.

```
# Initial calculations to evaluate prediction quality
Y_train_mean_cv = Y_train_cv.mean()
Y_train_meandev_cv = ((Y_train_cv-Y_train_mean_cv)**2).sum()
Y_test_meandev_cv = ((Y_test_cv-Y_train_mean_cv)**2).sum()

# initialize model
RForreg = RandomForestRegressor(max_depth=max_depth[depth],
                                n_estimators=n_estimators[
   ↪ number_trees],
                                        max_leaf_nodes=max_leaf_nodes[
   ↪ leaf],
                                        max_samples=max_samples[sample
   ↪ ],
                                        random_state=0)

# train the model
RForreg.fit(X_train_cv, Y_train_cv["cnt"])

# Make predictions based on the training set
Y_train_pred_cv = RForreg.predict(X_train_cv)
Y_train_dev_cv = ((Y_train_cv["cnt"]-Y_train_pred_cv)**2).sum
   ↪ ()
r2_cv = 1 - Y_train_dev_cv/Y_train_meandev_cv

# Evaluate the result by applying the model to the test set
Y_test_pred_cv = RForreg.predict(X_test_cv)
Y_test_dev_cv = ((Y_test_cv["cnt"]-Y_test_pred_cv)**2).sum()
pseudor2_cv = 1 - Y_test_dev_cv/Y_test_meandev_cv

# Append results to dataframe
new_row = {'R2': r2_cv,
            'PseudoR2': pseudor2_cv,
            'fold': folds[fold],
            'max_depth': max_depth[depth],
            'n_estimators': n_estimators[number_trees],
            'max_leaf_nodes': max_leaf_nodes[leaf],
            'max_samples': max_samples[sample]}
```

Code Excerpt 10: Cross validation using *TimeSeriesSplit* - part 2

Subsequently, the best combination of hyperparameters is determined. The mean of the Pseudo-R^2 across all folds is calculated for each combination. If the calculated mean exceeds the current maximum, the variable *index* is changed to the index of the current hyperparameter combination. At the end, *index* is used to retrieve the best hyperparameter combination and to store them in *best_parameters* for a final round of model training and testing. We employed Pseudo-R^2 instead of R^2 as a decision criterion for the best hyperparameters, so as to take into account the robustness of the model.

```
# Calculate means to find the best hyperparameters across all
   ↪ folds
n_folds = folds[-1]
i = 0
```

```
index = 0
mean_max = 0
while i < len(df_parameters):
    if df_parameters.iloc[i:i+n_folds, 0].mean() > mean_max:
        mean_max = df_parameters.iloc[i:i + n_folds, 0].mean()
        index = i
        i += n_folds
    else:
        i += n_folds
    df_parameters = df_parameters.append(new_row, ignore_index
  ↪ =True)

# best parameters based on mean of PseudoR^2
best_parameters = pd.Series(df_parameters.iloc[index, 3:])
```

Code Excerpt 11: Cross validation using *TimeSeriesSplit* - part 3

Refer to chapter 4 for the different hyper-parameter combinations used throughout cross validation for each model. The following table provides a summary about the splitting and cross validation strategies we applied for each of our models.

Data Partitioning - Summary of Splitting and Cross Validation Strategies		
	Initial Train-Test-Split	**Cross Validation Strategy**
1	Random Train-Test-Split	GridSearchCV (KFold)
2	Time-Based Train-Test-Split	TimeSeriesSplit
3	Time-Based Train-Test-Split	GridSearchCV (KFold)

4 Model Development

In the following section, the machine learning models that were developed in the course of this group project will be presented. Having taken a look at the dataset in the previous preprocessing steps, we were able to confidently conclude that the problem at hand was clearly non-linear, which is why linear models such as linear or ridge regression were left out from our model selection.

4.1 Scikit-Learn: RandomForestRegressor

One machine learning approach to solve a non-linear problem, such as the one at hand, is a random forest of regression trees [36]. Scikit-learns *RandomForestRegressor* is an ensemble method, i.e. a group of predictors [26, p. 183]. A random forest consists of multiple random decisions trees [37]. The decision trees are created by splitting the training set in two subsets using a single feature and a threshold, e.g. hour < 6.5. The feature and threshold that produce the purest subsets are chosen for the split. The resulting subsets are splitted recursively following the same logic until the maximum depth is reached or no further split can be found that reduces impurity [26, p. 171]. Each decision tree in the random forest predicts the amount of bikes rented for each hour. The final prediction of bikes rented is the average of the predictions of all individual decision trees. This averaging is implemented to control overfitting and enhance the prediction quality of the model [38].

4.1.1 Random Split and GridSearchCV Approach

For one, we created a *RandomForestRegressor* based on a random train-test-split (refer to chapter 3.6.1).

The first task while hyperparameter tuning was to gain an understanding about the effects of the relevant parameters of the *RandomForestRegressor* on the model's quality and robustness. We therefore focused on the parameters impact on overfitting and on computational complexity. This also included the understanding about the mechanics of the default settings.

To enhance our comprehension about the parameters' impact on R^2 and Pseudo-R^2, we splitted the training set once more into a train- and test-set for a preliminary analysis and visualization. Doing so enabled us to acquire an initial understanding about the behaviour of R^2 and Pseudo-R^2 depending on different values for each hyperparameter without using the originally created test-set (refer to chapter 3.6.1).

```
# Split Trainig set once more for an initial visualization of
    ↪ the impact of the hyperparameters to avoid overfitting
X_train_2, X_test_2, Y_train_2, Y_test_2 = train_test_split(
    ↪ X_train, Y_train, test_size = 0.2, random_state=0)
```

Code Excerpt 12: Further train-test-split for a preliminary analysis

First, we inspected *n_estimators*, which determines the number of trees in the forest [38]. Generally, the more trees a forest contains, the better the prediction quality. At the same time, the computation time increases linearly with each tree added to the forest [39]. Knowing that the results will stop improving significantly beyond a certain number of trees, it might not be reasonable to add more and more trees to the forest from an efficiency point of view. While increasing the number of trees will most likely not degrade or improve the model, it will definitely increase the computational complexity. As can be inferred from Figure 14, the R^2 and Pseudo-R^2 are only significantly increasing up to around 100 trees. Afterwards, the increases can be interpreted as negligible (if they're increasing at all).

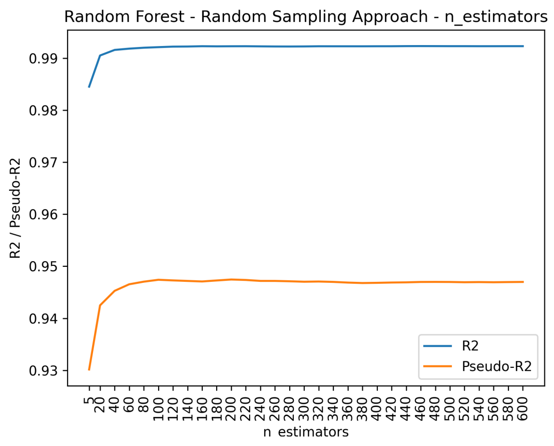


Figure 14: Preliminary analysis - random sampling approach - *n_estimators*

Secondly, we focused on *max_depth* and *max_leaf_nodes* as they both limit the tree size and thereby control overfitting [40]. *max_depth* determines the maximum depth of each tree while *max_leaf_nodes* sets a condition on the splitting of the nodes in the tree. Leaving both parameters at their default may lead to overfitting, because each tree would grow without restriction.

From Figure 15 can be deduced that a tendency of overfitting starts to occur from *max_depth* of around 8 and *max_leaf_nodes* of around 80 onward.

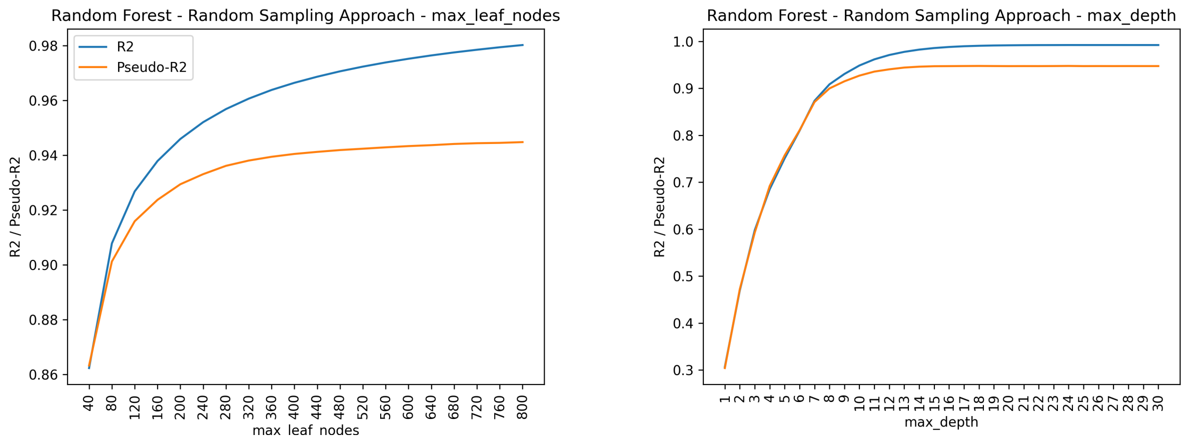


Figure 15: Preliminary analysis - random sampling approach - *max_leaf_nodes* & *max_depth*

Based on our preliminary analysis, we determined the ranges for the hyperparameters as follows:

```
param_grid = {
    'max_depth': [8, 9, 10, 11, 12, 13],
    'n_estimators': [80, 100, 120],
    'max_leaf_nodes': [60, 70, 80],
    'max_samples': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99],
}
```

Code Excerpt 13: Final parameters used for cross validation

Additionally to the previously mentioned parameters, *max_samples* was included. *max_samples* limits the size of the subsets [38], i.e. by determining what percentage of the entire data is allocated to each tree.

The following tables provide a summary of the final, non-default parameters and the metrics:

Hyperparameters	
n_estimators	120
max_depth	12
max_leaf_nodes	80
max_samples	0.3

Metrics	
R^2	0.9103
Pseudo-R^2	0.9001

While it may stand out that some of the final values for the hyperparameters are at the boundary of the respective ranges, we intentionally decided to do so. While an increase in *max_leaf_nodes* may for example lead to an increase R^2 and the Pseudo-R^2, it would also lead to a greater divergence between those two metrics thereby increasing the risk that the model is not stable. An increase in *n_estimators* will not, as previously outlined, significantly increase the models performance, but negatively impact its computational complexity. We therefore decided not to increase the values for *max_leaf_nodes* and *n_estimators* further.

4.1.2 Time-Based Split and TimeSeriesCV Approach

Additionally to a model based on a random train-test-split, we trained another model based on the concept of a random forest, but this time by splitting the data in line with the time series approach (refer to chapter 3.6.2). In terms of cross validation, we used *time-based splitting* to also fully consider the time series character of the given dataset (refer to chapter 3.6.3). The pursued approach for determining the hyperparameters and training the model is identical to the one outlined in chapter 4.1.1.

As a preliminary step, we examined the impact of the different hyperparameter combinations on R^2 and Pseudo-R^2 with the only difference that the additional train-test-split took into account the time-series-character of the dataset by using the function *train_test_split_ts* (refer to chapter 3.6.2).

Based on our preliminary analysis, we determined the ranges for the hyperparameters as follows and used them for cross validation using *TimeSeriesCV*:

```
max_depth =  [8, 9, 10, 11,
  ↪ 12]
n_estimators = [100, 120, 140]
max_leaf_nodes = [60, 70, 80]
max_samples = [0.1, 0.2, 0.3,
  ↪ 0.5, 0.6, 0.7, 0.8, 0.9,
  ↪  0.99]
```

Code Excerpt 14: Final parameters used for cross validation

The final, non-default hyperparameters as well as the corresponding metrics are summarized in the following:

Hyperparameters	
n_estimators	120
max_depth	10
max_leaf_nodes	80
max_samples	0.5

Metrics	
R^2	0.9089
Pseudo-R^2	0.8715

4.1.3 Time-Based Split and GridSearchCV Approach

Finally, we applied a train-test-split based on time (refer to chapter 3.6.2) while using *GridSearchCV* for cross validation. The preliminary analysis was performed in an identical way to the one outlined in chapter 4.1.2.

On the basis of our initial analysis, we specified the ranges for the hyperparameters as follows and applied them using scikit-learns *GridSearchCV*:

```
    'max_depth': [8, 9, 10,
↪  11, 12],
    'n_estimators': [100, 120,
↪   140],
    'max_leaf_nodes': [60, 70,
↪   80],
    'max_samples': [0.1, 0.2,
↪  0.3, 0.5, 0.6, 0.7, 0.8,
↪   0.9, 0.99],
```

Code Excerpt 15: Final parameters used for cross validation

The following tables encapsulate the final, non-default hyperparameters together with the metrics:

Hyperparameters	
n_estimators	100
max_depth	10
max_leaf_nodes	80
max_samples	0.2

Metrics	
R^2	0.9094
Pseudo-R^2	0.8678

4.2 CatBoost Regressor

The CatBoost Regressor is based on the approach of gradient boosting on decision trees and is an open source framework of the russian company Yandex. The approach of boosted decision trees is close to the random forest approach but instead of building many trees and using the majority vote, it builds many trees and each new tree endeavours to correct the previous trees' mistakes. However, boosted decision trees tend to overfit after a few iterations, as illustrated in Figure 16 [41].

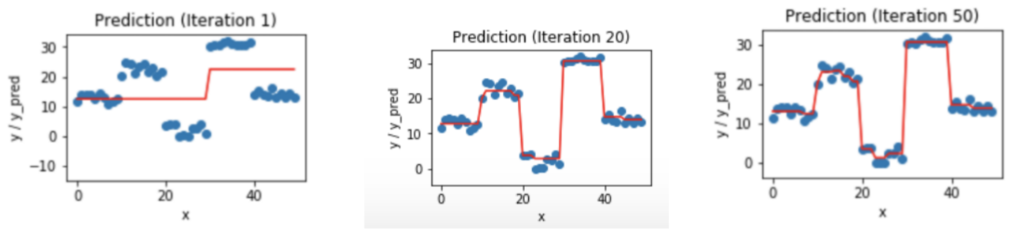


Figure 16: Gradient boosted decision tree after 1, 20, and 50 iterations

Beyond trying a new uncommon framework, the overfitting problem of gradient boosted decision trees was one of the primary reasons we decided to follow the gradient boosting approach with CatBoost.

CatBoost comes with five principal advantages over the Scikit-learn framework. Firstly, CatBoost is able to achieve great quality using the default parameters. Secondly, it supports categorical features which was a big advantage for our dataset because the CatBoost model was able to treat eight features of our dataset as categories. Another advantage is that CatBoost is able to train the model on GPU. Additionally, it is possible to apply the model very quickly and efficiently. Last but not least, CatBoost provides improved accuracy through the overfitting detector, which is easy to use and stops training the model if the improvement of each iteration is starting to stagnate [42].

The improvement of each iteration is measured by the loss function root mean squared error (RMSE) and is a parameter in the CatBoost regressor. The goal of the approach is to minimize the root mean squared error and to minimize the difference between R^2 and Pseudo-R^2. Furthermore, we had to optimize the parameters' iterations, learning rate and depth for the CatBoost regressor. The parameter iterations determines how often the gradient boosting tree is improved. The parameter learning rate sets the impact new iterations can have on the current model. At last, the parameter depth sets how many features the gradient boosted decision tree is allowed to consider. Additionally, we applied the two overfitting detector parameters *od_type* and *od_wait* for every model fitting.

After defining the parameters to consider when initializing the CatBoost model, we had to make some further albeit relatively minor changes. As mentioned, CatBoost is able to deal with categorical features. However, it is necessary to tell the model those features it should treat as categories. Therefore, we set the features *season*, *mnth*, *hr*, *holiday*, *weekday*, *workingday*, *wheathersit* and *rush_hour* as cat_features, which is the categorical parameter in CatBoost. We also discovered that the models performed slightly better if we casted the categorical features as an int64 datatype. We thought this observation was quite interesting, though we were unable to find an obvious explanation for it.

4.2.1 Random Split and GridSearchCV Approach

To train the first CatBoost model we used a random sampling approach. Before commencing the training, we applied all aforementioned settings and performed a *GridSearchCV* for the parameters' iterations, depth and learning rate. For the parameter iterations we tried 30, 50, 100, 200, 400, 600, 800 and 1000. Additionally, we tried the learning rates 0.01, 0.05, 0.1, 0.2 and 0.3 and at last we tried depths of 6, 8 and 10.

The *GridSearch* identified a depth of 10, a learning rate of 0.05 and 1000 iterations as the optimized model parameters. We implemented these parameters and trained the model on the randomly sampled dataset with the random state 0. The overfitting detector stopped training after 675 iterations. Afterwards we calculated the R^2 and Pseudo-R^2 and achieved convincing results. One advantage of this approach is the mixture of historic and current data points, allowing the model to recognize current trends in the data. As we predict bike rentals of the test-set, the model learned the autocorrelation between the previous and the subsequent hour-long periods as well (i.e. the previous and subsequent rows on the dataset).

Hyperparameters	
iterations	1000
depth	10
learning_rate	0.05

Metrics	
R^2	0.9574
Pseudo-R^2	0.9459

4.2.2 Time-Based Split and TimeSeriesCV Approach

The second CatBoost model was trained with our *TimeSeriesCV*. We were able to split the dataset in a fashion aligned with our time series. This approach is equivalent to that outlined in chapter 3.6.2. The *TimeSeriesCV* tried the same parameters as the random sampling approach, though the *TimeSeriesCV* calculated different optimal parameters for the CatBoost regressor. Using this approach we identified 200 iterations, a learning rate of 0.2 and a max depth of 6 as the model's optimized parameters.

Hyperparameters	
iterations	200
depth	6
learning_rate	0.2

Metrics	
R^2	0.9620
Pseudo-R^2	0.9002

4.2.3 Time-Based Split and GridSearchCV Approach

The third model was trained with a time series split such that the first 80% of the data was used to train the model while the last 20% of the dataset was used to test the model. As with our first approach, we performed a *GridSearchCV* to calculate the optimized parameters to train the model, since every approach has a different structured dataset and needs to be adapted accordingly. Therefore, we used a depth of 6, 1000 iterations and a learning rate of 0.1 to train our model. However, our overfitting detector stopped training after just 260 iterations. Based on this and on the R^2 and Pseudo-R^2, we can conclude that this approach tends to overfit quite easily. In contrast to our first approach, the 4% difference between the R^2 and Pseudo-R^2 is significantly higher. The explanation for this observation is that we are trying to predict future bike rentals based on historical data. By predicting bike rentals in the far future, the model is not trained on the current trends and the accuracy of the prediction decreases as a result.

Hyperparameters	
iterations	1000
depth	6
learning_rate	0.1

Metrics	
R^2	0.9437
Pseudo-R^2	0.9062

After applying all three different approaches, we are able to compare the R^2 and Pseudo-R^2 values. As we are able to see the the best trained and most robust CatBoost model is the model trained by random sampled data. As already mentioned this approach knows historic and current bike rental trends which improves the quality of the model. Nonetheless, this does not represent use cases of the real world, since we only have historic data to predict the future. In conclusion we wouldn't be able use such a model in the real world.

4.3 Scikit-Learn: SVR

Another machine learning approach that can be applied to classification and regression tasks as well as for both linear and nonlinear problems is the one used in support vector machines (SVMs) [43]. SVMs are based on the idea of identifying an optimal hyperplane based on given hyperparameters with its special characteristic being the implementation of a margin of tolerance regarding the inclusion of support vectors [44]. Non-linear functionality can be achieved via kernel functions transforming the data into higher dimensions before applying linear separation [45]. In the following section, the method of SVMs applied to the given regression problem will be referred to as support vector regression (SVR).

We used the Python machine learning library Scikit-learn and its SVR implementation with the main hyperparameters comprising *epsilon*, *kernel*, *degree*, *gamma* and *C*. *Epsilon* defines the margin of tolerance in which no penalty is given to errors. As mentioned before, *kernel* functions are used for increasing the level of dimension of the dataset in order to allow linear separations by the hyperplane to behave in a non-linear way with regards to the original data. Like we already discussed, the regression problem at hand is clearly non-linear, which is why a linear kernel function was not taken into account in our analysis. The non-linear functions radial basis function (RBF) and polynomial kernel function with its hyperparameter *degree* were specifically looked at for the identification of the optimal SVR models. In addition to the kernel functions, the SVR model also takes in *gamma* and *C* [46]. *C* is an L2 regularization parameter that trades off correct classification/regression against maximization of the function's margin. Higher *C* values lead to larger margins with increased number of training instances ignored for inclusion which can, in turn, mitigate overfitting. *Gamma* represents the kernel coefficient for the selected *kernel* function. In simple words, this means that it defines the degree of influence of each single training instance with high values meaning weak and low values meaning strong influence [47]. If, for example, the *gamma* value is really high, the radius of the area of influence around each support vector only includes itself which leaves no room for regularization via *C*. The process of identifying optimal hyperparameters and how they affect the final models will be covered for each approach in the following subsections.

4.3.1 Random Split and GridSearchCV Approach

The first approach followed to find a well-suited SVR model was based on random sampling (refer to chapter 3.6.1) paired with randomized cross-validation as a special case of *GridSearchCV* (refer to chapter 3.6.3). Due to the long processing durations of the calculation-heavy SVR, we opted for Scikit-learn's *RandomizedSearchCV* rather than the extensive *GridSearchCV*. *RandomizedSearchCV* is a cross-validation method for hyperparameter optimization that uses the same hyperparameter space as *GridSearchCV* but incorporates random hyperparameter combinations whose total number is less than all the combinations used in *GridSearchCV* [48]. While the performance might be only slightly worse, the heavily improved run time makes up for that [49]. In our case, we found that *RandomizedSearchCV* allowed us to conduct additional cross-validation iterations, which were more important for identifying optimal models than one single extensive *GridSearchCV* run.

Having done three iterations of *RandomizedSearchCV* with an initially broad range of hyperparameters (see here for extensive documentation on hyperparameters), several interesting observations were made by us. First, the RBF kernel function seemed to outperform the polynomial kernel function with its different degree values which is why we dropped the polynomial kernel function and the hyperparameter degree after the first iteration. Also, the margin of tolerance (*epsilon*) levelled out at a low value of 0.01 meaning that there is only a relatively small *epsilon*-tube where no penalty is associated with data instances in the training loss function. Based on an example provided by Scikit-learn [47] we know that we ended up with a medium *gamma* of 1 and a relatively low C of 1.75. If we use the other two upcoming SVR models for comparison, however, this C is significantly higher than their C values. If we now look at the R^2 value, we see that the resulting model achieved an accuracy on the training set of 0.9141 with a Pseudo-R^2 of 0.9167 which highlights the models robustness due to the absence of any over- or underfitting. In this case C might have mitigated overfitting through L2 regularization.

The following table summarizes the final, non-default parameters and R^2 and Pseudo-R^2:

Hyperparameters	
kernel	RBF
degree	-
gamma	1.0
epsilon	0.01
C	1.75

Metrics	
R^2	0.9141
Pseudo-R^2	0.9167

4.3.2 Time-Based Split and TimeSeriesCV Approach

Following the SVR model based on a random train/test split and *RandomizedSearchCV*, next, we split the data following a time series approach (refer to chapter 3.6.2) and also implemented a cross validation method based on chronological folds in order to fully account for the time series characteristic of the given data (refer to section 3.6.3). Due to the additional complexity introduced from the *TimeSeriesCV* we had to simplify the hyperparameter space by dropping the polynomial kernel and the degree hyperparameter from the start and only conducting one iteration of the extensive *TimeSeriesCV* with hyperparameters still kept broad with a slight orientation based on the optimal values for the other two approaches in order to keep up a high likelihood of achieving accurate results.

The optimal hyperparameters (see here for extensive documentation on hyperparameters) were a larger but still relatively small *epsilon* of 0.03 resulting in a narrow margin of tolerance, a significantly smaller C of 0.5 compared to the optimal hyperparameters based on 4.3.1) and a smaller *gamma* of 0.5. These hyperparameters still yield a high train set accuracy with an R^2 of 0.9353 but also exhibit significant overfitting with a Pseudo-R^2 of only 0.8632. The small epsilon value paired with low regularization contributed through C might be reasons for this noticeable overfitting gap between R^2 and Pseudo-R^2. Additional differences in splitting and cross-validating result in completely different conditions for the model to be trained upon. Specifically, if the model is trained on *TimeSeriesSplit* it takes into account older data to a higher degree due to being trained on a chronologically preceding training set which impedes the inclusion of the dataset's more recent patterns which could be newer trends of usage for example.

The following table summarizes the final, non-default parameters and R^2 and Pseudo-R^2:

Hyperparameters	
kernel	RBF
degree	-
gamma	0.5
epsilon	0.03
C	0.5

Metrics	
R^2	0.9353
Pseudo-R^2	0.8632

4.3.3 Time-Based Split and GridSearchCV Approach

Finally, we used time series-sensitive splitting (refer to section 3.6.2) in conjunction with *RandomizedSearchCV* as a special case of *GridSearchCV* (refer to chapter 3.6.3 and chapter 4.3.1 for advantages of *RandomizedSearchCV*).

After having done three iterations of *RandomizedSearchCV* (see here for extensive documentation on hyperparameters), we yielded optimal parameters that closely resemble the results from the previous splitting and cross-validation approach (refer to chapter 4.3.2) with an RBF *kernel* function, *gamma* of 0.6, *epsilon* of 0.03 and *C* of 0.5. The fact that the model achieved a similar accuracy with an R^2 of 0.9398 and Pseudo-R^2 of 0.8557 lets us derive that the impact of *TimeSeriesCV* vs. *GridSearchCV* is not that significant under these circumstances for an SVR model. However, the aforementioned conclusion that applying *time-based splitting* is likely to have caused this significant overfitting by giving higher importance to the chronologically preceeding data instances within the training set not accounting for changing patterns in towards or on the test set.

The following table summarizes the final, non-default parameters and R^2 and Pseudo-R^2:

Hyperparameters	
kernel	RBF
degree	-
gamma	0.6
epsilon	0.03
C	0.5

Metrics	
R^2	0.9398
Pseudo-R^2	0.8557

4.4 Scikit-Learn: MLPRegressor

Another important class of machine learning models comprises different types of artificial neural networks. They were started being introduced and researched in the middle of the last century and have become very popular lately as a central component of the deep learning field [50]. One of the most popular and more straightforward types of neural networks is the multilayer perceptron (MLP). It is a class of feedforward artificial neural networks comprising three different types of layers: one input layer, one output layer and at least one hidden layer. The layers' components are single neurons that take weighted inputs and aggregate them through the activation function leading to an output. During training, backpropagation is used for calculating weights and thereby optimizing the result of the loss function.

Again, we made use of the machine learning library Scikit-learn, specifically its *MLPRegressor* implementation [51]. An essential part of every neural network is the backpropagation approach defined by the specific solver that is used. While the solver can be included in the parameter grid to be optimized within the cross-validation, we decided from the beginning to use limited-memory BFGS due to its efficiency with datasets that are not considered big data (note that it does not use gradient descent). The other main hyperparameters include *hidden_layer_sizes*, alpha, and activation. As mentioned above, an MLP includes at least one hidden layer between the input and output layer. The hyperparameter *hidden_layer_sizes* defines both the number of hidden layers as well as the number of neurons in each of the layers thereby determining the structural organization and complexity of the neural network. In Scikit-learn's notation *hidden_layer_sizes* of (100, 100,) means 2 hidden layers with 100 neurons each while (25,) means only 1 hidden layer with 25 neurons.

The structure heavily impacts the neural network's capability of classifying and estimating. Research states that a neural network with a single hidden layer and finite number of neurons can approximate continuous functions with mild assumptions [52]. This might seem sufficient and questions the deployment of deep neural networks with deep referring to the number of hidden layers. However, the depth of the neural network also defines the way it learns and influences is ability to recognise hidden patterns in the input signals [53]. Regularization plays an important role in combating overfitting. The *MLPRegressor* includes an L2 penalty regularization parameter using *alpha*. Higher *alpha* values encourage smaller weights to be assigned that leads to decreased variance and overfitting [54]. Another essential hyperparameter is the *activation function* which aggregates the weighted input by using an activation function that determines the output that is sent to the next layer's neurons. Among the most popular activation functions there are sigmoid, tanh and ReLU. Their different characteristics like range on the y-axis and slope result in different behaviors of neural networks based on the selection of the activation function.

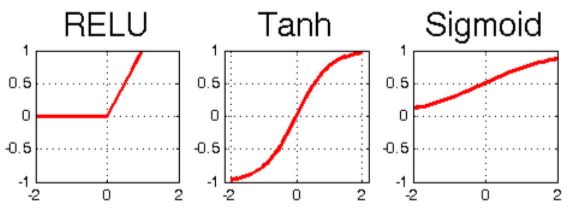


Figure 17: Plots of ReLU, Tanh, and Sigmoid activations sourced from [55]

4.4.1 Random Split and GridSearchCV Approach

Like for the other models, first, we used random sampling (refer to section 3.6.1) and randomized cross-validation (refer to section 3.6.3).

Initially, the hyperparameter grid to be searched for optimal values included a broad range of organizational structures with both one or two single layers and a total number of neurons of up to 300, all aforementioned activation functions and regularization parameters alpha between 0.01 and 0.3 (see here for extensive documentation on hyperparameters). Post-*RandomizedSearchCV* optimal hyperparameters defined the following resulting MLP: Shallow neural networks with only one hidden layer including 200 neurons using the ReLU activation function with an alpha of 0.025. Despite relatively low regularization via alpha [54], the model shows solid robustness with an R^2 of 0.9518 and Pseudo-R^2 of 0.9420 revealing a slight overfitting.

The following table summarizes the final, non-default parameters and R^2 and Pseudo-R^2:

Hyperparameters	
hidden_layer_sizes	(200,)
alpha	0.025
activation	ReLU

Metrics	
R^2	0.9518
Pseudo-R^2	0.9420

4.4.2 Time-Based Split and TimeSeriesCV Approach

In order to get a model that fully takes into account the time series nature of the dataset, we also identified optimal hyperparameters based on *time-based splitting* (refer to section 3.6.2) and *TimeSeriesCV* (refer to section 3.6.3). Since *TimeSeriesCV* conducts an extensive search over the full respective hyperparameter space given and neural networks require large amounts of computational resources, this might lead to long calculation times. In order to be able to optimze the hyperparameters efficiently we simplified the hyperparameters by reducing the total number of *hidden_layer_sizes* to be tested in order to be able to process a single iteration of *TimeSeriesCV* (see here for extensive documentation on hyperparameters).

In contrast to the previous model, hyperparameter optimization resulted in a completely different structure of the MLP with two hidden layers comprising of 50 and 25 neurons. Regularization was kept low at 0.015 with ReLU still being the optimal activation function. High accuracy on the training set with an R^2 of 0.9471 cannot be matched by the Pseudo-R^2 of 0.9014 resulting in a significant overfitting, which, again, might be traced back to the time-based splitting that stresses older data (for more extensive coverage on that refer to section 4.3.2).

The following table summarizes the final, non-default parameters and R^2 and Pseudo-R^2:

Hyperparameters	
hidden_layer_sizes	(50, 25,)
alpha	0.015
activation	ReLU

Metrics	
R^2	0.9471
Pseudo-R^2	0.9014

In Figure 18 the MLP structure for the models from 4.4.3 and 4.4.2 is shown:

4.4.3 Time-Based Split and GridSearchCV Approach

As a mixture of the two previously conducted approaches, we also analyzed models based on *time-based splitting* (refer to section 3.6.2) paired with *RandomizedSearchCV* as a special case of *GridSearchCV* (refer to section 3.6.3).

RandomizedSearchCV (see here for extensive documentation on hyperparameters) resulted in the same organizational structure as the previous approach defining an MLP with two hidden layers with 50 and 25 neurons each. Moreover, the alpha regularization was also similar at 0.02. Interestingly, this model uses the tanh rather than ReLU as an activation function (see activation functions in Figure 17). With regards to metrics, this MLP exhibits an R^2 of 0.9478 an a Pseudo-R^2 of 0.9203, decreasing the overfitting gap we saw in the previous model. This might be due to a slighltly increases strictness of regularization via *alpha* and/or to the difference in

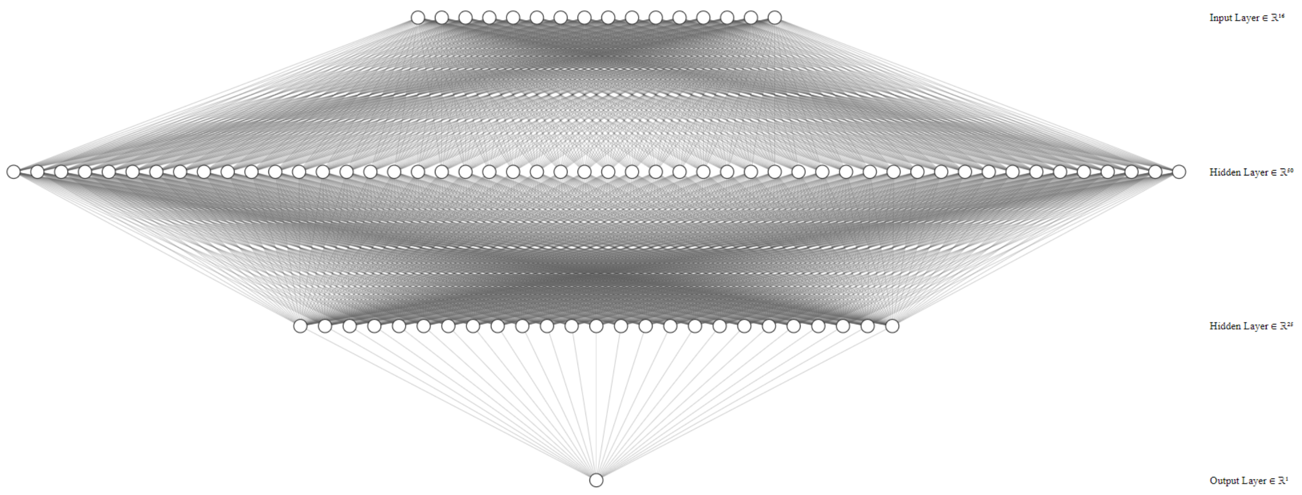


Figure 18: Structural organization of MLPs

TimeSeriesCV vs. *RandomizedSearchCV*.

Hyperparameters	
hidden_layer_sizes	(50, 25,)
alpha	0.02
activation	Tanh

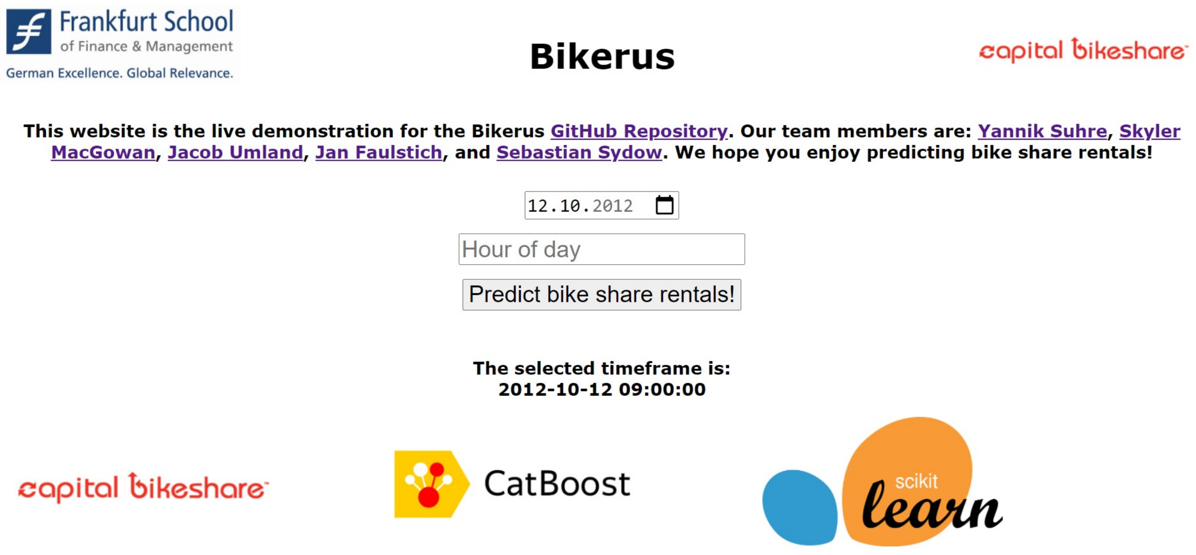
The following table summarizes the final, non-default parameters and R^2 and Pseudo-R^2:

Metrics	
R^2	0.9478
Pseudo-R^2	0.9203

5 Wrap-up

5.1 Applying our Models

All of the models we developed during the course of this project have been saved and loaded onto an app, from which live predictions can be effectuated. Users of this app choose a date and time of day during the period from which the test data was drawn, i.e. August 8 2012 - December 31 2012. The user then clicks ”Predict”, which causes the models to make a prediction for the specified time frame. In addition to the prediction outputted by each of the models, the user is also able to see the actual number of bike share rentals during the specified time frame. The app has been deployed and can be accessed here (loading this webpage can take some time!), so that you can try it yourself. Figure 19 below also provides an image of this app’s interface.



Bikerus

This website is the live demonstration for the Bikerus GitHub Repository. Our team members are: Yannik Suhre, Skyler MacGowan, Jacob Umland, Jan Faulstich, and Sebastian Sydow. We hope you enjoy predicting bike share rentals!

12.10.2012

Hour of day

Predict bike share rentals!

The selected timeframe is:
2012-10-12 09:00:00

Frankfurt School of Finance & Management
German Excellence. Global Relevance.
capital bikeshare
CatBoost
scikit learn

Figure 19: Bikerus demonstration

Should you run the *Bikerus* repository on your local machine, you can also access the *BikeRental* database with a tool like DBeaver and check the table *predicted_df*, which contains both the real count of rented bikes and the predicted number of bikes from all models developed during the course of this project. In order to start the app within the Docker container you have to enter the following into the bash.

```
python application.py
```

Code Excerpt 16: Start flask server

This starts the app. Overall, as manifested by the resources described above, our most effective model was the CatBoost regressor trained on randomly sampled data. Just based on the R^2 and Pseudo-R^2 values one could argue that the *MLPRegressor* of Scikit-learn trained on randomly sampled data is as good as the gradient boosted decision tree. However, we have to consider the huge amount of work we had to put in to encode the categorical features of the dataset in order for the *MLPRegressor* to obtain this result (as mentioned in 3.5, we had to encode five specific dataset features in the *MLPRegressor*). Conversely, the CatBoost regressor deals with categorical features quite easily; it did not require us to encode the categories. In sum, based on the CatBoost model's robustness (as reflected by its high R^2 and Pseudo-R^2 values) combined with the relatively light preprocessing effort it requires, we consider the first CatBoost model to be our most effective predictive model. However, we realize that this model does not represent the reality due to the data given. In the real world we can only use models trained on historic data.

5.2 Model Limitations

As discussed in the Introduction, bike sharing systems rely on the production of a large amount of high-quality data in order to work effectively. Hence, we were somewhat spoiled for this project in that the original dataset was of a high-quality to begin with, and required relatively little preprocessing and imputing. Still, the dataset was not an exhaustive list of all possible variables that could influence the demand for bike share rentals, and as with all models the output of our models is only as good as the data they were trained on. So, though we were lucky to have quite good quality data for this project, it wasn't "perfect" either so there is some degree of limitation in that regard.

Additionally, the accuracy of our models' predictions declines with an increasing difference between the desired prediction time and the time of our dataset (2011 & 2012). This is because our models were trained exclusively on data obtained during 2011 & 2012; they do not incorporate novel information when making a prediction. To make a reliable prediction for a time period other than sometime during 2011 or 2012, the models would need to be retrained using data obtained during the desired time frame.

As outlined in chapter 3.4.1, all of the machine learning models presented in this report predict the aggregated number of bikes rented for a specific hour without differentiating between casual and registered users. Furthermore, the models return a single value as a prediction for the bikes rented across all rental stations. Developing machine learning models for each of these two customer types and/or predictions for each rental station is an area of potential (model) enhancement that could provide additional benefits to the rental company. For example, casual users might predominately use bikes close to tourist attractions or recreational areas whereas registered users might mostly use the bikes when commuting to and from work. Differences in user behaviour would impact the demand at each rental station, depending on its specific location as illustrated in Figure 1. Predictions regarding the bikes rented from each customer type and/or at each location would enable the rental company to make better use of the bikes in stock by deploying them to the different rental stations accordingly. It should be noted that making predictions about the bikes rented at each rental station would require more granular data relative to that provided in the original dataset.

Another factor limiting our models' accuracy is that due to constraints in computational resources and time we had to use somewhat simplified approaches during model optimization. Especially for resource-heavy models such as SVR or MLP, we leveraged *RandomizedSearchCV* and simplified hyperparameter grid inputs, sacrificing some degree of model accuracy to help facilitate model development. Both of these approaches reduce the number of cross-validated hyperparameter combinations, which might lead to missing out on the optimal composition of hyperparameters.

5.3 Key Takeaways

To conclude this paper, we would like to reflect on our team's main takeaways from this project, especially as they relate to group work in a data analytics and machine learning context.

For one, having the same software foundation is a critical component of ensuring efficacious group collaboration. Because all of our team members used the same software foundation, we never ran into issues wherein we were not able to run/incorporate alterations made by other team members. This culminated in what was ultimately a fairly simple deployment of our prediction app to azure webservices.

Another key takeaway was settling on the right approach; establishing a general group *modus operandi.* At the start of the project there were a variety of approaches put forward to such tasks as researching, testing, and writing. Fairly early on however we established general parameters regarding how we would go about these tasks, which helped us collaborate in an efficient manner.

Finally, while we all learn a little differently, when developing new skills such as programming there is no doubt that practical experience working directly with the technology is paramount, and should be emphasized even from the relatively early stages of one's "learning journey". Doing so in a group context is especially helpful as it allows team members to learn from each other whilst directly applying their newfound knowledge.

References

[1] F. Siddiqui, “Lyft gets into bike-share business, acquiring operator of capital bikeshare and citi bike,” *The Washington Post*, 02.07.2018. [Online]. Available: https://www.washingtonpost.com/news/dr-gridlock/wp/2018/07/02/lyft-gets-into-bike-share-business-acquiring-operator-of-capital-bikeshare-and-citi-bike/

[2] R. van der Zee, “The growth of bike-sharing schemes around the world,” *The Guardian*, 26.04.2016. [Online]. Available: https://medium.com/@miathaole/the-growth-of-bike-sharing-schemes-around-the-world-696752074697

[3] Groen7.nl, “De geschiedenis van de witkar,” 2012. [Online]. Available: https://www.groen7.nl/de-geschiedenis-van-de-witkar/

[4] L. Thao, “The growth of bike-sharing schemes around the world,” 2019. [Online]. Available: https://medium.com/@miathaole/the-growth-of-bike-sharing-schemes-around-the-world-696752074697

[5] Market Watch, “Bike sharing market 2020 by global countries data, opportunities, competitive landscape industry size and growth by forecast to 2026 |absolute reports,” *MarketWatch*, 09.09.2020.

[6] J. Grudin, “Ai and hci: Two fields divided by a common focus,” *AI Magazine*, vol. 30, no. 4, p. 48, 2010.

[7] A. X. Zhang, M. Muller, and D. Wang, “How do data science workers collaborate? roles, workflows, and tools,” 2020. [Online]. Available: https://arxiv.org/pdf/2001.06684

[8] M. Dancho, “Part 3 - docker for data scientists: (a top skill for 2020),” 2019. [Online]. Available: https://www.business-science.io/business/2019/11/22/docker-for-data-science.html

[9] IBM, “The benefits of containerization and what it means for you,” 06.02.2019. [Online]. Available: https://www.ibm.com/cloud/blog/the-benefits-of-containerization-and-what-it-means-for-you

[10] J. Chiu, “Why your company needs python for business analytics,” 03.12.2019. [Online]. Available: https://www.datacamp.com/community/blog/why-your-company-needs-python-for-business-analytics

[11] D. Subramanian, “Docker — containerization for data scientists - towards ai - medium,” *Towards AI*, 22.05.2020. [Online]. Available: https://medium.com/towards-artificial-intelligence/docker-container-and-data-scientist-bae208ce8268

[12] DevopsCube, “What is docker? how does it work?” 2020. [Online]. Available: https://devopscube.com/what-is-docker/

[13] Microsoft, “Remote - containers - visual studio marketplace,” 19.12.2020. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-containers

[14] SQLite, “Release history of sqlite,” 18.12.2020. [Online]. Available: https://sqlite.org/changes.html

[15] ——, “About sqlite,” 18.12.2020. [Online]. Available: https://www.sqlite.org/about.html

[16] GWDG, “Overleaf,” 15.10.2020. [Online]. Available: https://info.gwdg.de/docs/doku.php?id=de:services:email_collaboration:sharelatex

[17] J. Walkenhorst, “How to interpolate time series data in python pandas,” 11.06.2019. [Online]. Available: https://towardsdatascience.com/how-to-interpolate-time-series-data-in-apache-spark-and-python-pandas-part-1-pandas-cff54d76a2

[18] C.-h. Chen, W. K. Härdle, and A. Unwin, *Handbook of data visualization.* Springer Science & Business Media, 2007.

[19] W. Atlas, “Monthly weather forecast and climate washington, dc,” n.d. [Online]. Available: https://www.weather-us.com/en/district-of-columbia-usa/washington-climate

[20] U. Jaitley, “Why data normalization is necessary for machine learning models,” 08.10.2018. [Online]. Available: https://medium.com/@urvashilluniya/why-data-normalization-is-necessary-for-machine-learning-models-681b65a05029

[21] scikit learn, “sklearn.preprocessing.minmaxscaler,” n. d. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

[22] J. Brownlee, “3 ways to encode categorical variables for deep learning,” 27.08.2020. [Online]. Available: https://machinelearningmastery.com/how-to-prepare-categorical-data-for-deep-learning-in-python/

[23] I. London, “Encoding cyclical continuous features - 24-hour time,” 31.07.2016. [Online]. Available: https://ianlondon.github.io/blog/encoding-cyclical-features-24hour-time/

[24] S. Anand, “Dummy variable trap,” 13.04.2019. [Online]. Available: https://medium.com/datadriveninvestor/dummy-variable-trap-c6d4a387f10a

[25] scikit learn, “3.1.2.1. cross-validation iterators for i.i.d. data,” n. d. [Online]. Available: https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation-iterators-for-i-i-d-data

[26] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 2nd ed. 1005 Gravenstein Highway North, Sebastopol, CA: O'Reilly Media, Inc., 2017.

[27] A. Bronshtein, "Train/test split and cross validation in python," 17.05.2017. [Online]. Available: https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6

[28] S. Srinidhi, "How to split your dataset to train and test datasets using scikit learn," 30.07.2018. [Online]. Available: https://contactsunny.medium.com/how-to-split-your-dataset-to-train-and-test-datasets-using-scikit-learn-e7cf6eb5e0d

[29] scikit learn, "sklearn.model_selection.train_test_split," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[30] D. Shulga, "5 reasons why you should use cross-validation in your data science projects," 27.09.2018. [Online]. Available: https://towardsdatascience.com/5-reasons-why-you-should-use-cross-validation-in-your-data-science-project-8163311a1e79

[31] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning, Second Edition: Data Mining, Inference, and Prediction*, 2nd ed., ser. Springer Series in Statistics. Berlin: Springer New York and Springer Berlin, 2009.

[32] M. Sanjay, "Why and how to cross validate a model?" 13.11.2018. [Online]. Available: https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f

[33] scikit learn, "sklearn.model_selection.gridsearchcv," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[34] ——, "3.1.2.5.1. time series split," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/cross_validation.html#time-series-split

[35] S. Wu, "3 best metrics to evaluate regression model?" 13.11.2018. [Online]. Available: https://towardsdatascience.com/what-are-the-best-metrics-to-evaluate-your-regression-model-418ca481755b

[36] J. Brownlee, "Machine learning algorithms mini-course," 29.04.2016. [Online]. Available: https://machinelearningmastery.com/machine-learning-algorithms-mini-course/

[37] H. Deng, "An introduction to random forest," 07.12.2018. [Online]. Available: https://towardsdatascience.com/random-forest-3a55c3aca46d

[38] scikit learn, "sklearn.ensemble.randomforestregressor," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

[39] ——, "1.11.2.3. parameters," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/ensemble.html#controlling-the-tree-size

[40] ——, "1.11.4.4. controlling the tree size," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/ensemble.html#controlling-the-tree-size

[41] P. Grover, "Gradient boosting from scratch," 9.12.2017. [Online]. Available: https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d

[42] Yandex, "Catboost - open-source gradient boosting library," 2020. [Online]. Available: https://catboost.ai/

[43] V. Kecman, "Support vector machines–an introduction," in *Support vector machines: theory and applications*. Springer, 2005, pp. 1–47.

[44] N. Bambrick, "Support vector machines: A simple explanation," 01.07.2016. [Online]. Available: https://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html

[45] M. K. Gurucharan, "Machine learning basics: Support vector regression," 11.07.2020. [Online]. Available: https://towardsdatascience.com/machine-learning-basics-support-vector-regression-660306ac5226

[46] scikit learn, "sklearn.svm.svr," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

[47] ——, "Rbf svm parameters," n. d. [Online]. Available: https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

[48] P. Worcester, "A comparison of grid search and randomized search using scikit learn," 05.06.2019. [Online]. Available: https://blog.usejournal.com/a-comparison-of-grid-search-and-randomized-search-using-scikit-learn-29823179bc85

[49] scikit learn, "Comparing randomized search and grid search for hyperparameter estimation," n. d. [Online]. Available: https://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html

[50] L. Hardesty, "Explained: Neural networks," *MIT News*, 14.04.2017. [Online]. Available: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

[51] scikit learn, "sklearn.neural_network.mlpregressor," n. d. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

[52] J. Heaton, "The number of hidden layers," 01.06.2017. [Online]. Available: https://www.heatonresearch.com/2017/06/01/hidden-layers.html

[53] N. Koleva, "When and when not to use deep learning," 01.05.2020. [Online]. Available: https://blog.dataiku.com/when-and-when-not-to-use-deep-learning

[54] scikit learn, “sklearn.neural_network.mlpregressor,” n. d. [Online]. Available: https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_alpha.html

[55] M. K. Hamdan, “Vhdl auto-generation tool for optimized hardware acceleration of convolutional neural networks on fpga (vgt),” 2018.