DSML Project Seoul Bike Data

January 6, 2025

1 Analysis of bike sharing demands in Seoul

1.1 Introduction

This project focuses on analyzing the factors influencing bike rentals for a bike-sharing company. Specifically, we aim to investigate whether temperature has a significant impact on the number of bikes rented. The dataset contains information from the years 2017 and 2018, including various variables such as daily temperature, weather conditions, and historical bike rental data. Our goal is to use this data to build a predictive model that can estimate the number of bikes that will be rented for each month. The model will help the company optimize bike availability and manage resources more efficiently based on temperature forecasts and other environmental factors

The dataset contains the count of public bicycles rented per hour in the Seoul Bike Sharing System, with corresponding weather data and holiday information. The dataset contains weather information (Temperature, Humidity, Windspeed, Visibility, Dewpoint, Solar radiation, Snowfall, Rainfall), the number of bikes rented per hour and date information. Each coloumn in "SeoulBikeData.csv" have the following description:

```
Date : the date of the day
Rented Bike Count : number of rented bike
Hour : the time in hour
Temperature(°C) : the temperature at this time
Humidity(%) : the pourcentage of humidity at this time
Wind speed (m/s) : the wind speed in m/s at this time
Visibility (10m) : the visibility per 10m at this time
Dew point temperature(°C) : the dew point temperature of day
Solar Radiation (MJ/m2) : the solar radiation in MJ/m2 at this time
Rainfall(mm) : the rain in mm at this time
Snowfall (cm) : the snowfalls in cm at this time
Seasons : the season of the day
Holiday : if the day is in holiday or not
Functioning Day : if rental company was working that day
```

1.2 Business Understanding

The city of Seoul is well organized for bikes, like cites in Danmark, and there is a good organisation for renting bikes. As so, the bikes renting data, from a company, have been collected and put in a database called "SeoulBikeData.csv". This dataset contain the number of rented bike every hour from 2017 to 2019 and allows us to see the cycles of bike renting every year. In fact, with those data the companies might want to forecast the bike renting, compared to the weather conditions,

the day or the season. The purpose of the project will be to design a historic model that could help the bike companies to deal with the bike stock with the wether forcast and the period. The following research question (RQ) havebeen formulated:

"Determine the optimum number of bikes needed each time of day based on hour, temperature and solar radiation."

PS : This Jupyter Notebook follows the chronological structure of the Data Science and Machine Learning course. It was our main tool for working on this course. That's why we trained multiple models and compared different techniques that were taught. We wanted to use this project to improve our skills in this field.

1.3 Data collection

The dataset that we use is from UC Irvine Machine Learning repository. It is composed of 14 columns (the features) and 8760 rows (the data). We choose this dataset because it can be useful in real life for real business and to see if its possible to forecast the number of rented bike. First in this part we are going to see how is the dataset and which feature are useful.

1.3.1 The Library

Libraries are crucial for expanding Python's capabilities, improving efficiency, and offering solutions for a broad spectrum of tasks. To get started, it's necessary to import the required libraries

[1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from sklearn.model_selection import LeaveOneOut, cross_val_score
from statsmodels.stats.outliers_influence import variance_inflation_factor as␣
 ↪VIF
from sklearn.metrics import mean_squared_error, accuracy_score, r2_score,␣
 ↪mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.neighbors import NearestNeighbors, KNeighborsClassifier,␣
 ↪KNeighborsRegressor
from sklearn.model_selection import LeaveOneOut, cross_val_score
from ISLP import load_data
from ISLP.models import (ModelSpec as MS, summarize, poly)
from sklearn.linear_model import LinearRegression
%matplotlib inline
plt.style.use('fivethirtyeight')
```

Pandas library enable to managing and preparing the data.
Numpy library contribute to handling numerical operations.
Matplotlib library contribute to better data visualization.
Seaborn library is employed to create visualizations of statistical data.

Sklearn library is employed for machine learning and modelling.

[2]:
```
df = pd.read_csv('SeoulBikeData.csv', encoding='unicode_escape')
```

1.4 Data Cleaning and Data Preparation

After introducing the different libraries to the program it is possible to import the single dataset at disposal. The dataset has 8760 rows and 14 columns, later referred to as features.

Data cleaning and data manipulation are necessary steps before take a closer look at the data

Further data cleaning and exploration are shown as to separate steps. In some cases exploration needs to happen while doing data cleaning because it can become an iterative process. Data cleaning is a crucial step in any data science and machine learning project. It contains of identification and adjustment of data quality issues, which can significantly impact the accuracy and reliability of further analysis and models development.

[3]:
```
df.info() #looking globaly at the dataset
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8760 entries, 0 to 8759
Data columns (total 14 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Date                       8760 non-null   object
 1   Rented Bike Count          8760 non-null   int64
 2   Hour                       8760 non-null   int64
 3   Temperature(°C)            8760 non-null   float64
 4   Humidity(%)                8760 non-null   int64
 5   Wind speed (m/s)           8760 non-null   float64
 6   Visibility (10m)           8760 non-null   int64
 7   Dew point temperature(°C)  8760 non-null   float64
 8   Solar Radiation (MJ/m2)    8760 non-null   float64
 9   Rainfall(mm)               8760 non-null   float64
 10  Snowfall (cm)              8760 non-null   float64
 11  Seasons                    8760 non-null   object
 12  Holiday                    8760 non-null   object
 13  Functioning Day            8760 non-null   object
dtypes: float64(6), int64(4), object(4)
memory usage: 958.3+ KB
```

[4]:
```
df.shape
```

[4]:
```
(8760, 14)
```

[5]:
```
df.isna().sum()
```

[5]:
```
Date                       0
Rented Bike Count          0
```

```
Hour                          0
Temperature(°C)               0
Humidity(%)                   0
Wind speed (m/s)              0
Visibility (10m)              0
Dew point temperature(°C)     0
Solar Radiation (MJ/m2)       0
Rainfall(mm)                  0
Snowfall (cm)                 0
Seasons                       0
Holiday                       0
Functioning Day               0
dtype: int64
```

[6]:
```
df.duplicated().sum()
```

[6]:
```
0
```

Through our analysis it is possible to see that the dataset doesn't contain any categorical data for each feature analyzed.

Furtherome the inevstigation of duplicates led to the conclusion that duplicated cells nor values are included in the original dataframe. As a conclusion number of rows and columns have remained the same.

Missing data can arise for various reasons, such as incomplete records or data entry errors. By examining these columns, a deeper understanding is gained of the dataset's characteristics. A identification of which columns have missing values and assess whether these gaps can be filled with reasonable imputations. In our specific case no missing values were reported.

[7]:
```
df.describe().T
```

[7]:

	count	mean	std	min	25%	\
Rented Bike Count	8760.0	704.602055	644.997468	0.0	191.00	
Hour	8760.0	11.500000	6.922582	0.0	5.75	
Temperature(°C)	8760.0	12.882922	11.944825	-17.8	3.50	
Humidity(%)	8760.0	58.226256	20.362413	0.0	42.00	
Wind speed (m/s)	8760.0	1.724909	1.036300	0.0	0.90	
Visibility (10m)	8760.0	1436.825799	608.298712	27.0	940.00	
Dew point temperature(°C)	8760.0	4.073813	13.060369	-30.6	-4.70	
Solar Radiation (MJ/m2)	8760.0	0.569111	0.868746	0.0	0.00	
Rainfall(mm)	8760.0	0.148687	1.128193	0.0	0.00	
Snowfall (cm)	8760.0	0.075068	0.436746	0.0	0.00	

	50%	75%	max
Rented Bike Count	504.50	1065.25	3556.00
Hour	11.50	17.25	23.00
Temperature(°C)	13.70	22.50	39.40

Humidity(%)	57.00	74.00	98.00
Wind speed (m/s)	1.50	2.30	7.40
Visibility (10m)	1698.00	2000.00	2000.00
Dew point temperature(°C)	5.10	14.80	27.20
Solar Radiation (MJ/m2)	0.01	0.93	3.52
Rainfall(mm)	0.00	0.00	35.00
Snowfall (cm)	0.00	0.00	8.80

1.5 Data Manipulation

Data manipulation can be crucial in order to make data exploration phase easier. In our case, the main temporal feature was the date. To simplify data analysis and exploration, additional features were created, including days of the week, months, and seasons of the year. As a result, three new columns—and therefore three new features—were added to the existing 14 features in the database, bringing the total to 17 features. Additionally in this step units from column names have been removed, together with blank spaces and uppercase letters. On top of that 3 features has been encoded and Functioning Day feature has been completely dropped as on non-functioning day there were no bike rentals.

[8]:
```
#Removing all of the useless charatere to simplify the column titles
df.columns = df.columns.str.replace(r"\s*\(.*?\)\s*", "", regex=True)
df.columns = df.columns.str.replace(" ", "_", regex=False)
```

[9]:
```
df.columns = [x.lower() for x in df.columns]
```

[10]:
```
df["date"] = pd.to_datetime(df["date"], dayfirst = True)
df["day"] = df['date'].dt.day
df["month"] = df['date'].dt.month
df["year"] = df['date'].dt.year
df["weekday"] = df['date'].dt.day_name()
```

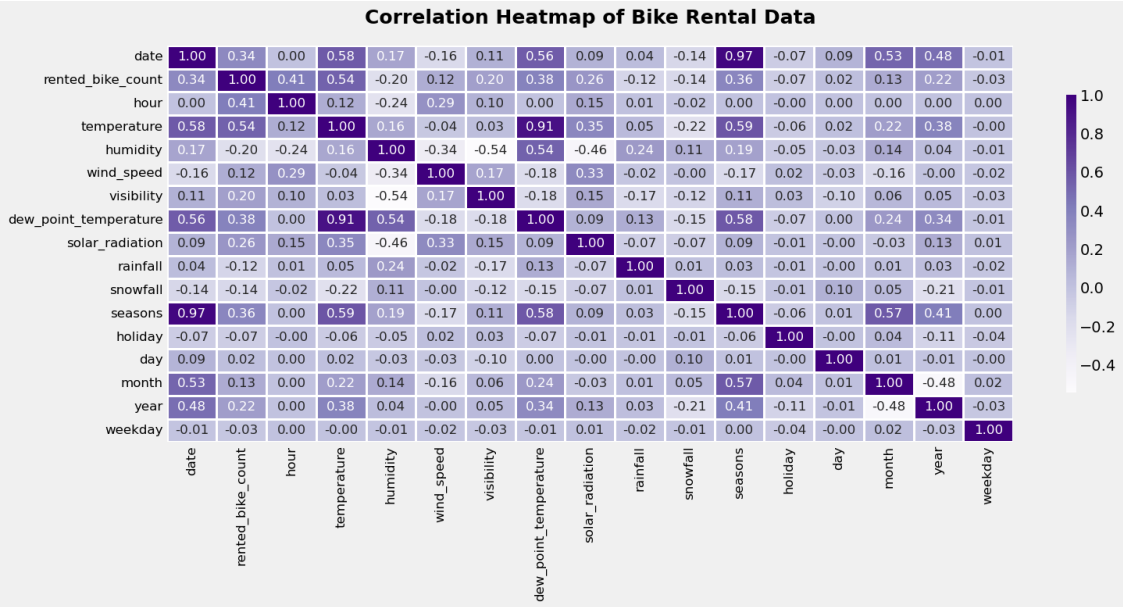
[11]:
```
df['seasons'] = df['seasons'].map({'Winter': 0, 'Spring': 1, 'Summer': 2,␣
↪'Autumn': 3})
df['holiday'] = df['holiday'].map({"No Holiday": 0, "Holiday": 1})
df['weekday'] = df['weekday'].map({"Monday": 1, "Tuesday": 2, "Wednesday": 3,␣
↪"Thursday": 4, "Friday": 5, "Saturday": 6, "Sunday": 7})
```

[12]:
```
df=df.drop(['functioning_day'], axis = 1)
```

1.6 Heat Map

In the following section, the heat map is introduced in order to discover correlations between the numerical features. The heat map shows that there is a correlation between some features: The main feature to analyze “rented bike count” shows acceptable and relevant level of correlations with seasonality, temperature, dew point temperature (referred to the temperature at which the external air became dry –> strictly linked to humidity).

[13]:
```
fig, ax = plt.subplots(figsize=(16, 8))
sns.heatmap(df.corr(),
            annot=True,
            fmt='1.2f',
            annot_kws={'size': 12},
            linewidths=1,
            linecolor='white',
            cmap='Purples',
            cbar_kws={"shrink": 0.75, "aspect": 30})

plt.title('Correlation Heatmap of Bike Rental Data', fontsize=18,␣
 ↪fontweight='bold', pad=20)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```

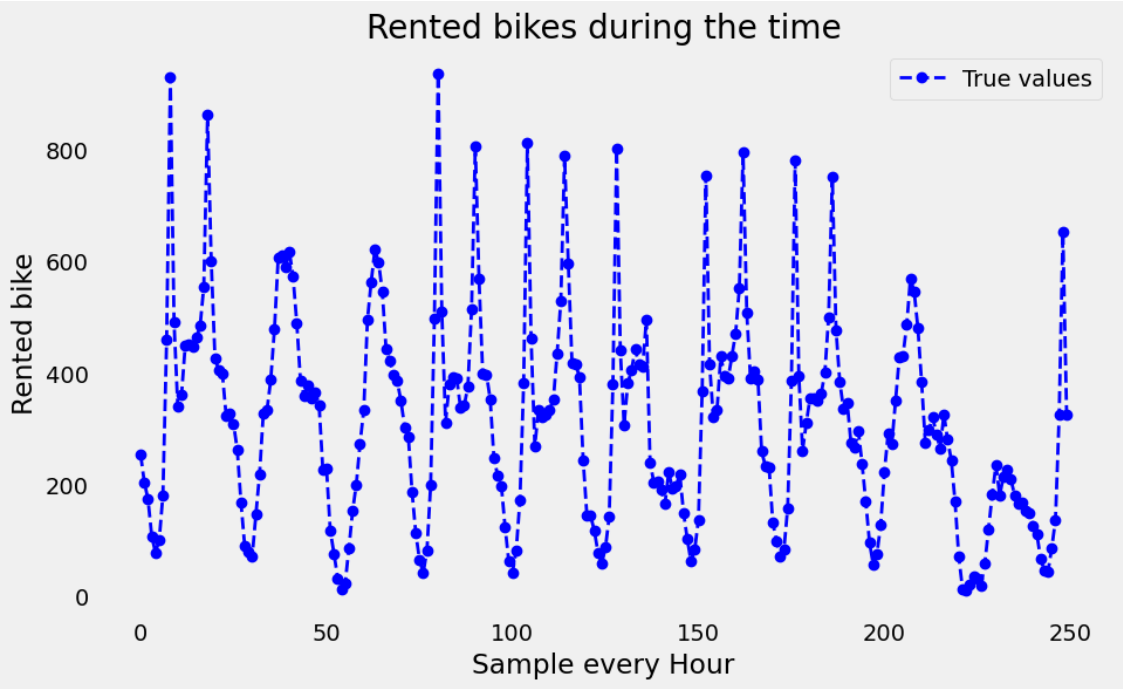


1.7 Data Exploration

In this section, we conduct a thorough exploration of the dataset, employing visualization techniques and examining correlations between features to gain a deeper understanding.

[14]:
```
#Plot of the rented bike per hour to visualise the dataset and the the␣
 ↪different trends
#We use the matplotlib library to plot figure
y= df['rented_bike_count'] #rented_bike is the part that we
plt.figure(figsize=(10, 6))
```

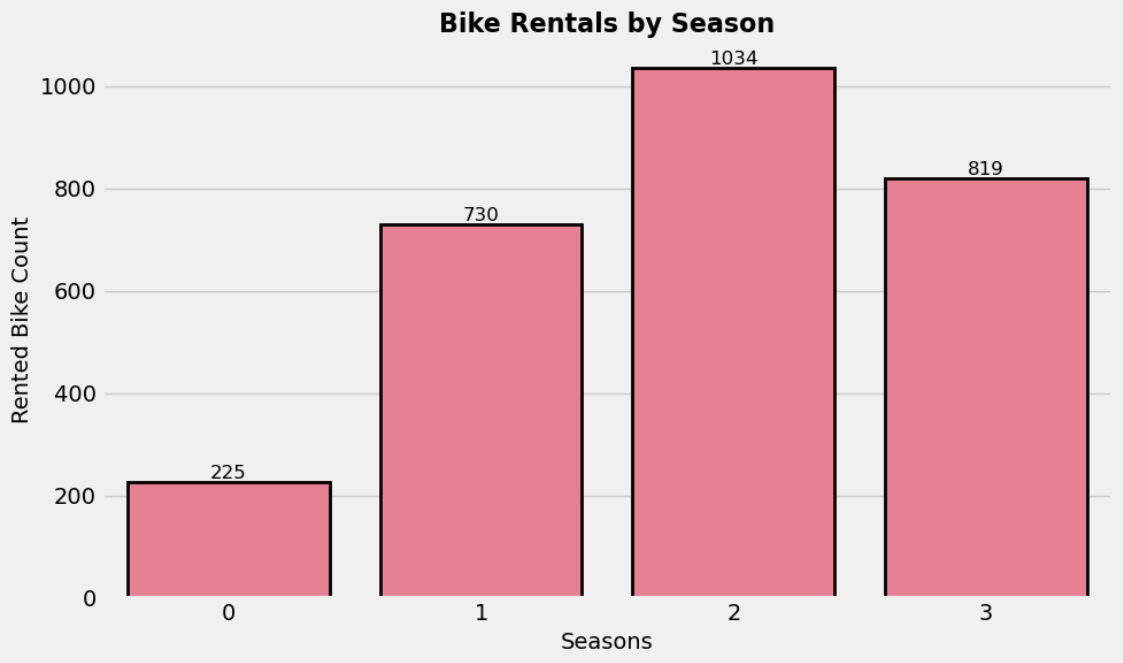
```
plt.plot(y[0:250], 'o', label='True values',␣
 ↪color='blue',linestyle='dashed',linewidth=2)

plt.title('Rented bikes during the time')
plt.xlabel('Sample every Hour')
plt.ylabel('Rented bike')
plt.legend()
plt.grid()
plt.show()
```



This graph shows the number of bikes rented over time. Sampling is done every hour. The data used is the actual data from the dataset to see the apparent trend. We can clearly see a daily or even weekly trend. On working days, we can see peaks in bike hire when people are going to and from work. We can also see that these peaks disappear at the weekend. Typically, however, we can see that despite the apparent trends, rentals vary a great deal and therefore depend not only on the day but also on other external parameters such as the weather. All the data collected will allow us to understand and model these trends but also to predict the number of bikes rented according to external parameters.

[15]:
```
plt.figure(figsize=(10, 6))
sns.set_palette("husl")

ax = sns.barplot(x="seasons", y="rented_bike_count", data=df, errorbar=None)

plt.title("Bike Rentals by Season", fontsize=16, fontweight='bold')
```

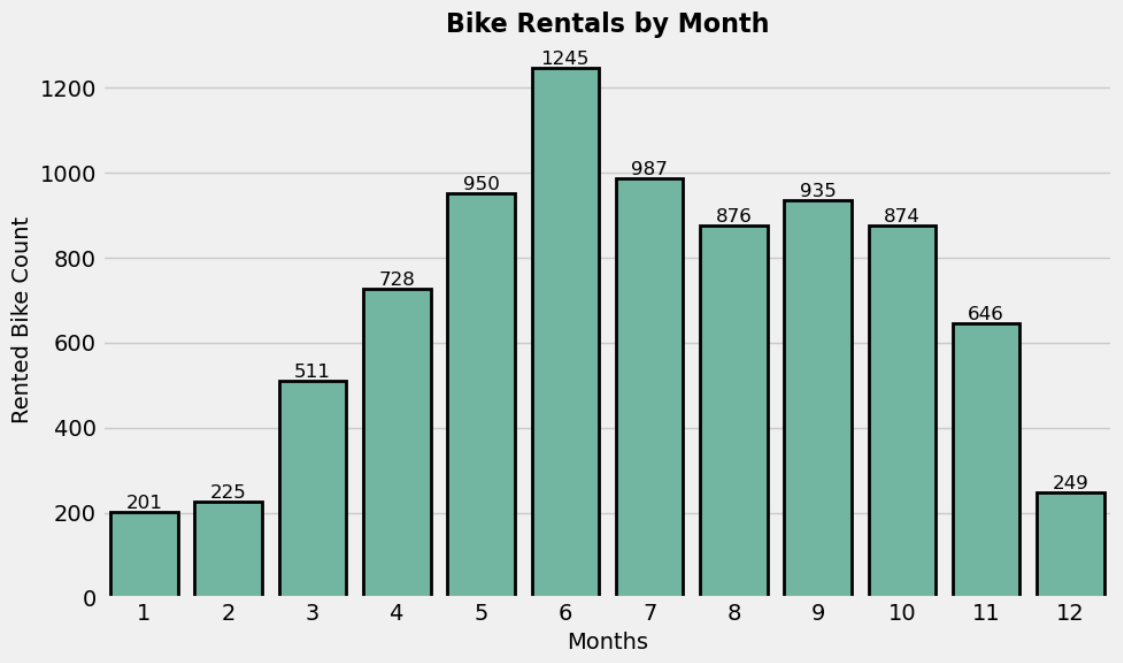
```
plt.xlabel("Seasons", fontsize=14)
plt.ylabel("Rented Bike Count", fontsize=14)

for patch in ax.patches:
    patch.set_edgecolor('black')
    patch.set_linewidth(2)

for p in ax.patches:
    ax.annotate(f'{int(p.get_height())}', (p.get_x() + p.get_width() / 2, p.
 ↪get_height()),
                ha='center', va='bottom', fontsize=12, color='black')

plt.tight_layout()
plt.show()
```



Starting from visualization of the number of bike rented per season through years and then breakdown the same variables per month and by hour of the day, this is displayed that mainly during summer and autumn the peak of rented bike is reached due to favourable weather conditions as confirmed by plot referred to bike rentals by month, highlighting that starting from May until October (peak in June) rental bike services are utilized.

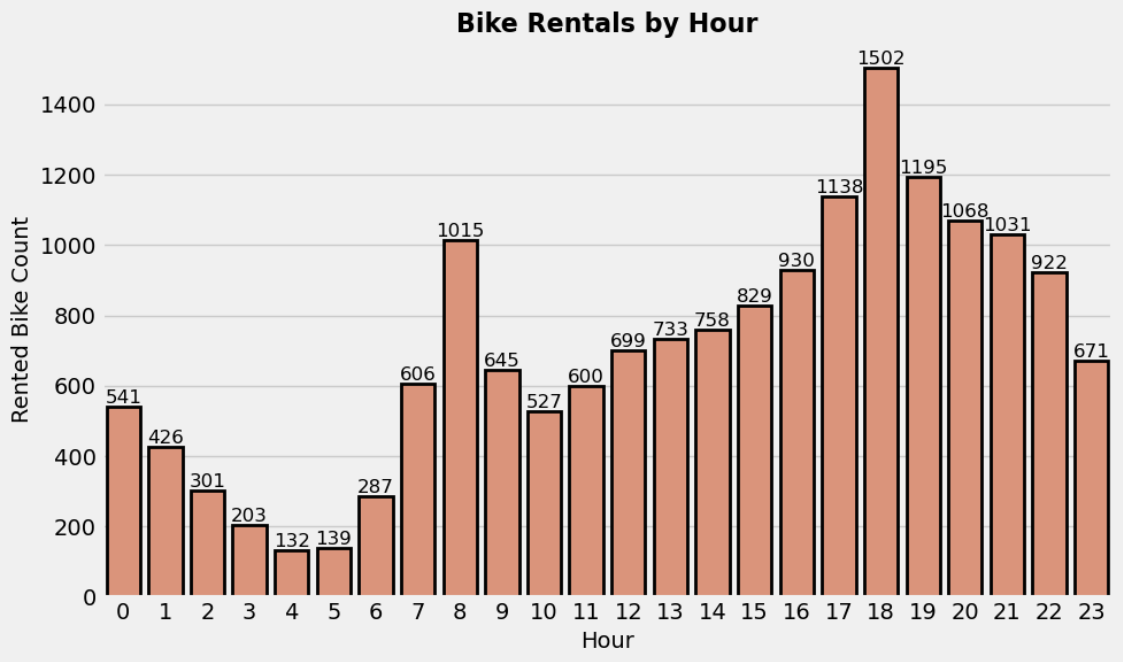
[16]:
```
plt.figure(figsize=(10, 6))
sns.set_palette("Set2")

ax = sns.barplot(x="month", y="rented_bike_count", data=df, errorbar=None)
```

```
plt.title("Bike Rentals by Month", fontsize=16, fontweight='bold')
plt.xlabel("Months", fontsize=14)
plt.ylabel("Rented Bike Count", fontsize=14)

for patch in ax.patches:
    patch.set_edgecolor('black')
    patch.set_linewidth(2)

for p in ax.patches:
    ax.annotate(f'{int(p.get_height())}', (p.get_x() + p.get_width() / 2, p.
 ↪get_height()),
                ha='center', va='bottom', fontsize=12, color='black')

plt.tight_layout()
plt.show()
```



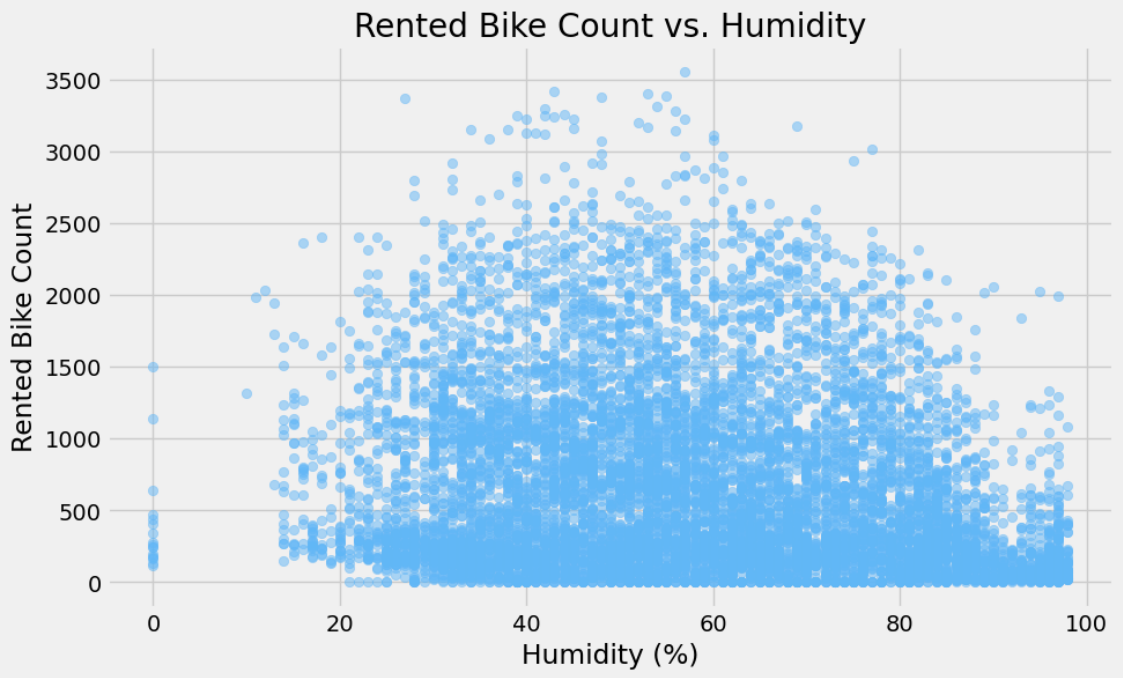
[17]:
```
plt.figure(figsize=(10, 6))
sns.set_palette("flare")

ax = sns.barplot(x="hour", y="rented_bike_count", data=df, errorbar=None)

plt.title("Bike Rentals by Hour", fontsize=16, fontweight='bold')
plt.xlabel("Hour", fontsize=14)
plt.ylabel("Rented Bike Count", fontsize=14)
```

```
for patch in ax.patches:
    patch.set_edgecolor('black')
    patch.set_linewidth(2)

for p in ax.patches:
    ax.annotate(f'{int(p.get_height())}', (p.get_x() + p.get_width() / 2, p.
 ↪get_height()),
                ha='center', va='bottom', fontsize=12, color='black')

plt.tight_layout()
plt.show()
```

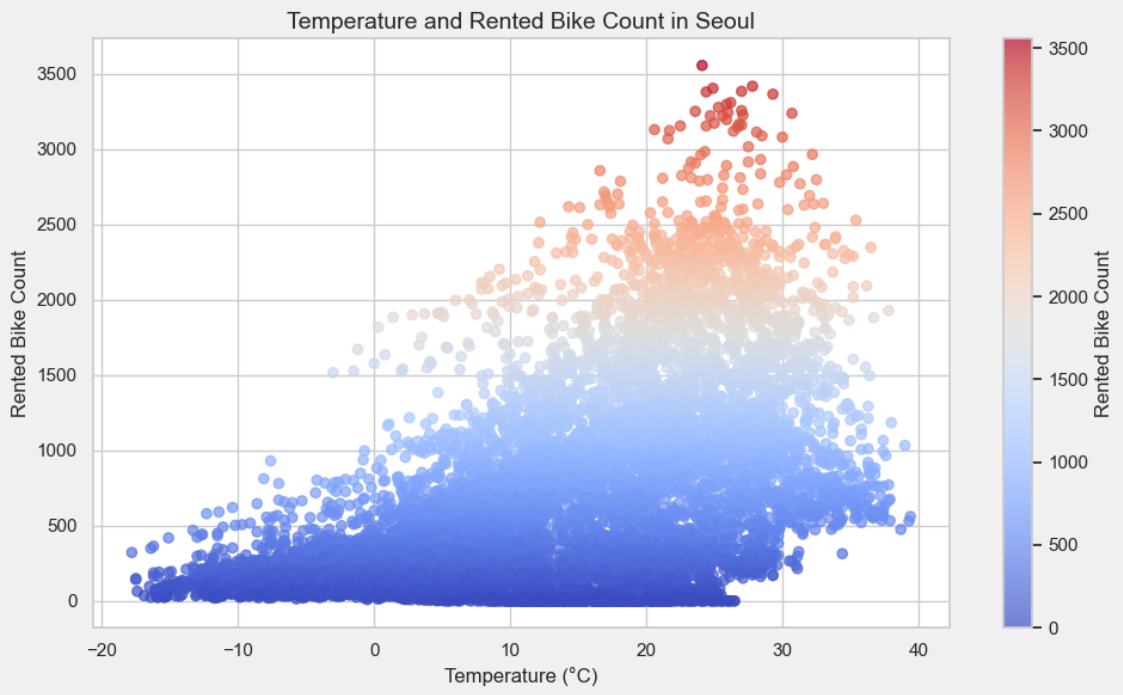


While the rented bike per hour can help understand the most likely reasons are for commuting home and work and hang out/ moving around the city during evening/dinner/after dinner.

[18]:
```
plt.figure(figsize=(10, 6))
plt.scatter(df['humidity'], df['rented_bike_count'], c="#61b7f7", alpha=0.5)
plt.title('Rented Bike Count vs. Humidity')
plt.xlabel('Humidity (%)')
plt.ylabel('Rented Bike Count')
plt.grid(True)
plt.show()
```



[19]:
```
plt.figure(figsize=(10, 6))
sns.set(style="whitegrid")

scatter = plt.scatter(x=df['temperature'], y=df['rented_bike_count'],
                      c=df['rented_bike_count'], cmap='coolwarm', alpha=0.7)

plt.title('Temperature and Rented Bike Count in Seoul', fontsize=14)
plt.xlabel('Temperature (°C)', fontsize=12)
plt.ylabel('Rented Bike Count', fontsize=12)

plt.colorbar(scatter, label='Rented Bike Count')

plt.tight_layout()
plt.show()
```

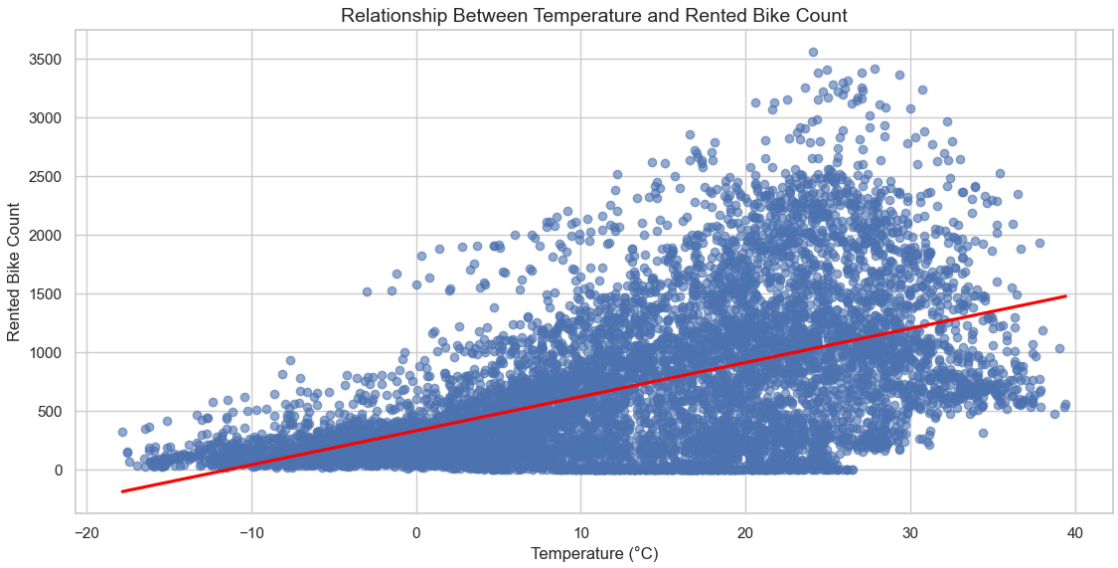


1.8 Ralation between the temperature and the Rented bikes

In this part we wanted to see the regression of the rented bikes and only the temperature as a parameter. The method we used is already in the seaborn library and it will put the linear regression of the rented bike with temperature.

[20]:
```
sns.set_theme(style="whitegrid")
plt.figure(figsize=(12, 6))

sns.regplot(
    x=df['temperature'], y=df['rented_bike_count'],
    scatter_kws={'alpha': 0.6},
    line_kws={'color': 'red'},
    ci=None
)

plt.title('Relationship Between Temperature and Rented Bike Count', fontsize=14)
plt.xlabel('Temperature (°C)', fontsize=12)
plt.ylabel('Rented Bike Count', fontsize=12)
```

[20]:
```
Text(0, 0.5, 'Rented Bike Count')
```



[21]:
```
# Define the data frame manualy
X = pd.DataFrame({'intercept': np.ones(df.shape[0]), 'temperature':␣
 ↪df['temperature']})
X[:5]
```

[21]:

	intercept	temperature
0	1.0	-5.2
1	1.0	-5.5
2	1.0	-6.0
3	1.0	-6.2
4	1.0	-6.0

[22]:
```
y= df['rented_bike_count']
model = sm.OLS(y,X) # does not fit the model, but specifies it
results = model.fit()#this fit the model with the data that we input the line␣
 ↪before
```

[23]:
```
results.summary()
```

[23]:

Dep. Variable:	rented_bike_count	**R-squared:**	0.290
Model:	OLS	**Adj. R-squared:**	0.290
Method:	Least Squares	**F-statistic:**	3578.
Date:	Mon, 06 Jan 2025	**Prob (F-statistic):**	0.00
Time:	10:56:25	**Log-Likelihood:**	-67600.
No. Observations:	8760	**AIC:**	1.352e+05
Df Residuals:	8758	**BIC:**	1.352e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P> |t|	[0.025	0.975]
intercept	329.9525	8.541	38.631	0.000	313.210	346.695
temperature	29.0811	0.486	59.816	0.000	28.128	30.034

Omnibus:	954.681	**Durbin-Watson:**	0.271
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1421.965
Skew:	0.817	**Prob(JB):**	1.68e-309
Kurtosis:	4.108	**Cond. No.**	25.9

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In this part we are creating new data to see the result and how it is functioning, we are going to do some predictions with those data

[24]:
```
new_df = pd.DataFrame({'intercept': np.ones(3), 'temperature': [1,6,-9]}) #new␣
↪data to see the prediction
new_df #data visualisation
```

[24]:

	intercept	temperature
0	1.0	1
1	1.0	6
2	1.0	-9

[25]:
```
new_predictions = results.get_prediction(new_df) #get the prediction of the␣
↪model that we created before
```

[26]:
```
new_predictions.predicted_mean #We use the mean because the new predictions are␣
↪presented as intervals
```

[26]:
```
array([359.03361294, 504.43910789,  68.22262305])
```

[27]:
```
# Produce confidence intervals for the predicted values:
new_predictions.conf_int(alpha=0.05)
```

[27]:
```
array([[342.97691266, 375.09031322],
       [491.30143052, 517.57678526],
       [ 44.46360153,  91.98164457]])
```

[28]:
```
# Prediction intervals are computed by setting obs=True:
new_predictions.conf_int(obs=True, alpha=0.05)
```

[28]:
```
array([[-706.47185251, 1424.53907839],
       [-561.02636685, 1569.90458262],
       [-997.42674254, 1133.87198864]])
```

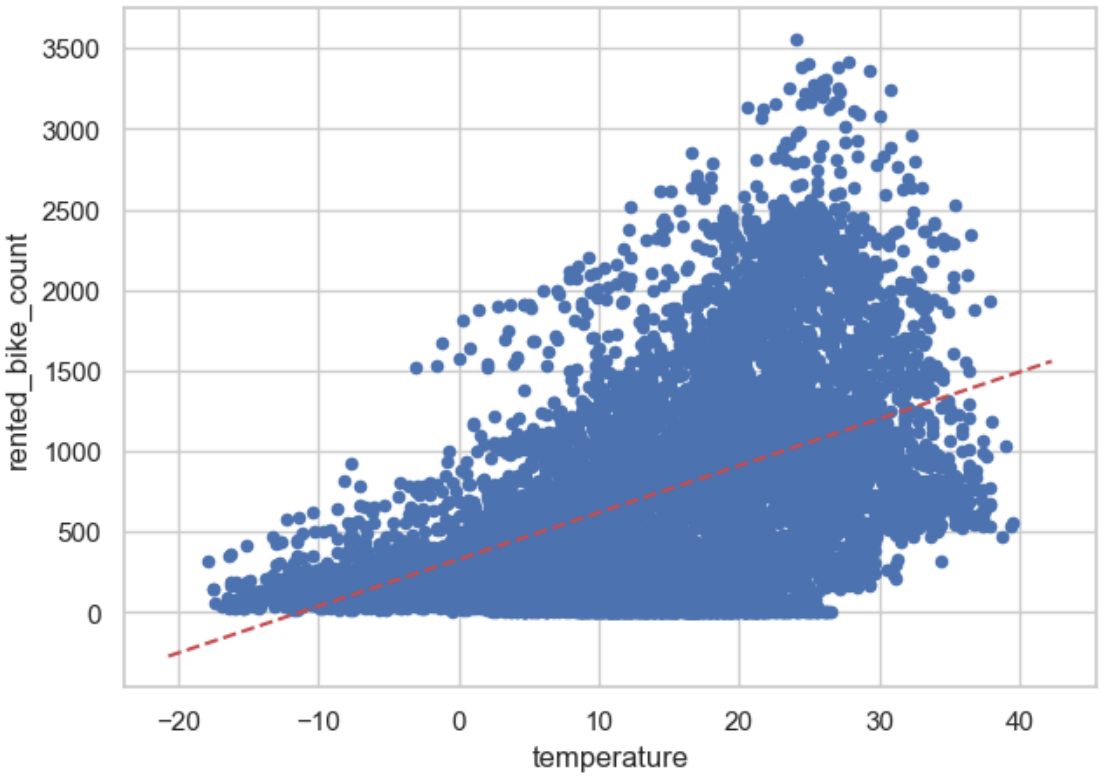
[29]:
```
def abline(ax, b, m): # defining the function
    "Add a line with slope m and intercept b to ax"
    xlim = ax.get_xlim()
```

```
    ylim = [m*xlim[0] +b, m*xlim[1] +b]
    ax.plot(xlim, ylim)
```

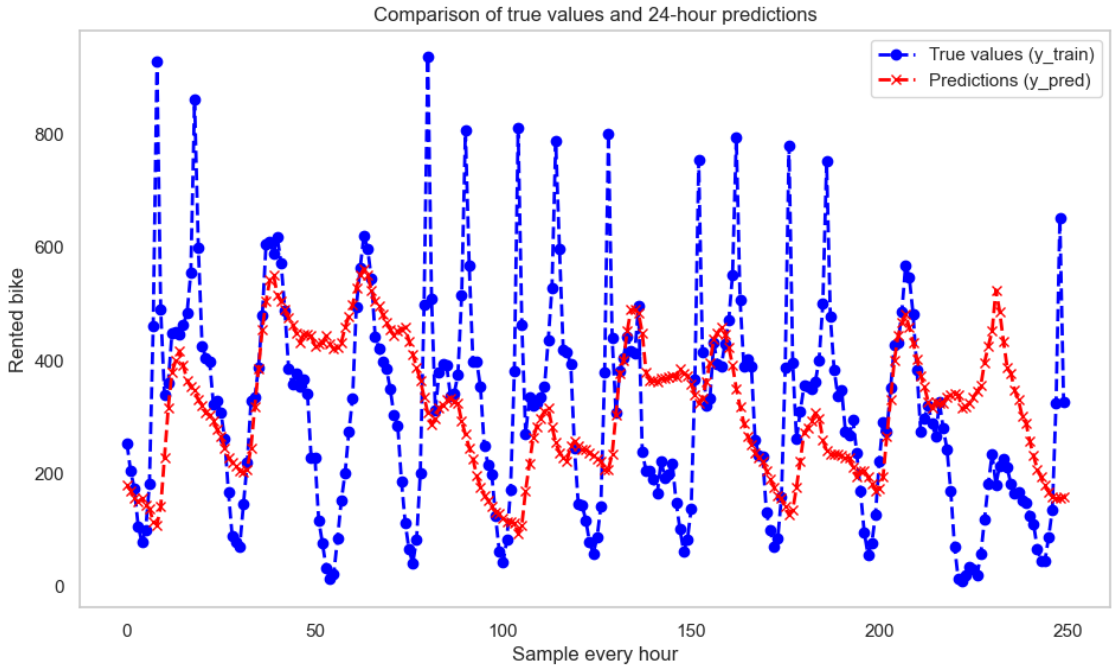
[30]:
```
# Including additional arguments: *args allows a number of non-named arguments␣
 ↪to abline
def abline(ax,b,m, *args, **kwargs): # **kwards allows any number of named␣
 ↪arguments, e.g., linewith=3 to abiline
    "Add a line with slope m and intercept b to ax"
    xlim = ax.get_xlim()
    ylim = [m *xlim[0]+b,m*xlim[1]+b]
    ax.plot(xlim, ylim, *args, **kwargs)
```

[31]:
```
# Let's use the new function and add the regression line to the plot of medv vs.
 ↪ lstat:
ax = df.plot.scatter('temperature', 'rented_bike_count')
abline(ax,
      results.params[0],
      results.params[1],
      'r--')#, # produces a red dashed line
      #linewith=3) # should define the line width
```

```
C:\Users\Raphael Preis\AppData\Local\Temp\ipykernel_12616\615515267.py:4:
FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a
future version, integer keys will always be treated as labels (consistent with
DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  results.params[0],
C:\Users\Raphael Preis\AppData\Local\Temp\ipykernel_12616\615515267.py:5:
FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a
future version, integer keys will always be treated as labels (consistent with
DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  results.params[1],
```



[32]:
```
y_pred_1 = results.get_prediction(X)

plt.figure(figsize=(10, 6))
plt.plot( y[0:250], 'o', label='True values (y_train)', color='blue'␣
 ↪,linestyle='dashed',linewidth=2)
plt.plot(y_pred_1.predicted_mean[0:250], 'x', label='Predictions (y_pred)',␣
 ↪color='red',linestyle='dashed',linewidth=2)

plt.title('Comparison of true values and 24-hour predictions')
plt.xlabel('Sample every hour')
plt.ylabel('Rented bike')
plt.legend()
plt.grid()
plt.show()
```



We can clearly see that there is not only one feature that impact the number of rented bikes. But this model showed us that there is a daily tendence.

On the graph, we can see that the temperature is not linear, other parameters have to be in the regression to get more precision in the model, due to this fact there is a lot of uncertainity in forecasting the number of rented bike.

1.9 MODELS AND FINDING THE FEATURE

The purpose of the following section will be to find the most impactful features to create the best model that would have the best forecast of the rented bike. First, we try to create a multilinear regression with all the feature. The model that we are going to implement will be use the Ordinary Least Squared Method (OLS) from the sklearn library. So we're going to try it out with several parameters entered, starting with a random choice of parameter, and observe the model's predictions.

1.10 Train a multiple linear regression model

[33]:
```
#First Test we try with the regular OLS with some variables of the data set to␣
 ↪see if we have a better accuracy
# we are adding new feature
X = pd.DataFrame({'intercept': np.ones(df.shape[0]), 'temperature':␣
 ↪df['temperature'], 'humidity' : df['humidity'], 'rain': df['rainfall'],␣
 ↪'hour' : df['hour'], 'weekday': df['weekday'] })
model_2pred = sm.OLS(y,X)
results_2pred = model_2pred.fit()
```

```
results_2pred.summary(model_2pred)
```

[33]:

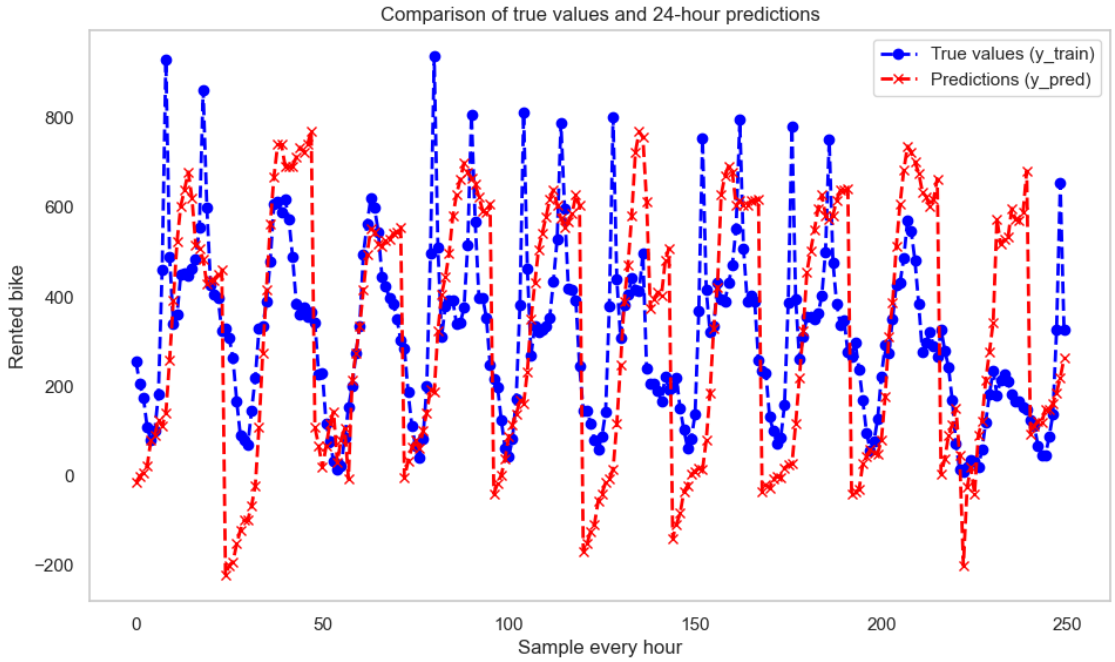
Dep. Variable:	<statsmodels.regression.linear_model.OLS object at 0x000002520D7EA930>		**R-squa**
Model:	OLS		**Adj. R**
Method:	Least Squares		**F-stati**
Date:	Mon, 06 Jan 2025		**Prob (**
Time:	10:56:26		**Log-Li**
No. Observations:	8760		**AIC:**
Df Residuals:	8754		**BIC:**
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P> |t|	[0.025	0.975]
intercept	405.5598	22.417	18.092	0.000	361.618	449.501
temperature	28.9787	0.434	66.698	0.000	28.127	29.830
humidity	-5.9385	0.269	-22.114	0.000	-6.465	-5.412
rain	-62.2572	4.618	-13.482	0.000	-71.309	-53.205
hour	27.8863	0.765	36.476	0.000	26.388	29.385
weekday	-9.9793	2.527	-3.948	0.000	-14.934	-5.025

Omnibus:	1024.159	**Durbin-Watson:**	0.444
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1951.978
Skew:	0.761	**Prob(JB):**	0.00
Kurtosis:	4.741	**Cond. No.**	285.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[34]:
```
#plotting the new multi linear model
y_pred_1 = results_2pred.get_prediction(X)
plt.figure(figsize=(10, 6))
plt.plot( y[0:250], 'o', label='True values (y_train)', color='blue'␣
 ↪,linestyle='dashed',linewidth=2)
plt.plot(y_pred_1.predicted_mean[0:250], 'x', label='Predictions (y_pred)',␣
 ↪color='red',linestyle='dashed',linewidth=2)


plt.title('Comparison of true values and 24-hour predictions')
plt.xlabel('Sample every hour')
plt.ylabel('Rented bike')
plt.legend()
plt.grid()
plt.show()
```



1.11 Comment :

We can see that the model has better quality then previously because it better interprets daily cycles. We can also see that it is getting closer and closer to the real values, but it is still not optimal. We also have a problem with this model, which is that due to linear regression it returns negative results, which poses a problem and cannot be realistic given that we are talking about bike rent. The model presented above has an R2 score of 0.46. The chosen model is therefore not very accurate but has better precision than the model with only one parameter, which has a score of 0.29. So we now need to choose the right input parameters to find the best model with linear regression.

1.12 Different Multi-linear Regression

1. Train and refine multiple linear regression models

First, a multiple linear regression model is trained with all 15 variables of the data set **(Model 1)**.

[35]:
```
# MODEL 1 (all 15 variables)
data_1 = df.columns.drop(['rented_bike_count', 'date'])
X_1 = MS(data_1).fit_transform(df)
y_1 = pd.DataFrame({'rented_bike_count': df['rented_bike_count']})
model_1 = sm.OLS(y_1, X_1)
results_1 = model_1.fit()
# results_1.summary(model_1)
print('Results model 1 (all 15 variables)')
print('R-squared:', results_1.rsquared.round(4))
```

```
print('Adj. R-squared:', results_1.rsquared_adj.round(4))
print('F-statistic:', results_1.fvalue.round(4))
print('AIC:', results_1.aic.round(4))
print('---------------------------------')
print('Values for all variables:')
print(results_1.pvalues.round(4))
```

```
Results model 1 (all 15 variables)
R-squared: 0.4868
Adj. R-squared: 0.486
F-statistic: 553.0188
AIC: 132387.7341
---------------------------------
Values for all variables:
intercept                0.0000
hour                     0.0000
temperature              0.0000
humidity                 0.0000
wind_speed               0.0065
visibility               0.7403
dew_point_temperature    0.3646
solar_radiation          0.0000
rainfall                 0.0000
snowfall                 0.0443
seasons                  0.0000
holiday                  0.0000
day                      0.6416
month                    0.0000
year                     0.0000
weekday                  0.0000
dtype: float64
```

Model 1 (all 15 variables) results in an R^2 of 0.487 and can therefore be classified as a model with insufficient accuracy. In addition, some variables (e.g. visibility, dew_point_temperature and day) have very high P-values. These variables therefore have no significant influence on the target variable.
In **Model 2**, we only use variables that are considered significant with regard to the correlation to rented_bike_count. This also reduces the model complexity.

[36]:
```
# Identifying significant features based on correlation to rented_bike_count
corr = df.corr()
features = corr["rented_bike_count"]
significant_features = features[features.abs() > 0.199]
significant_features
```

[36]:
```
date                   0.340772
rented_bike_count      1.000000
hour                   0.410257
```

```
temperature               0.538558
humidity                 -0.199780
visibility                0.199280
dew_point_temperature     0.379788
solar_radiation           0.261837
seasons                   0.359687
year                      0.215162
Name: rented_bike_count, dtype: float64
```

[37]:
```
data_2 = df.columns.drop(['rented_bike_count', 'date', 'wind_speed',␣
 ↪'rainfall', 'snowfall', 'holiday', 'day', 'month', 'weekday'])
X_2 = MS(data_2).fit_transform(df)
y_2 = pd.DataFrame({'rented_bike_count': df['rented_bike_count']})
model_2 = sm.OLS(y_2, X_2)
results_2 = model_2.fit()
# results_2.summary(model_2)
print('Results model 2 (8 significant variables)')
print('R-squared:', results_2.rsquared.round(4))
print('Adj. R-squared:', results_2.rsquared_adj.round(4))
print('F-statistic:', results_2.fvalue.round(4))
print('AIC:', results_2.aic.round(4))
print('---------------------------------')
print('Values for all variables:')
print(results_2.pvalues.round(4))
```

```
Results model 2 (8 significant variables)
R-squared: 0.469
Adj. R-squared: 0.4685
F-statistic: 966.275
AIC: 132672.461
---------------------------------
Values for all variables:
intercept                0.5333
hour                     0.0000
temperature              0.0000
humidity                 0.0000
visibility               0.9035
dew_point_temperature    0.0123
solar_radiation          0.0000
seasons                  0.0000
year                     0.5448
dtype: float64
```

With R^2 of 0.469, **Model 2** has a lower R^2 value than Model 1. The adjusted R^2, which takes model complexity into account and can be considered a fairer measure of model quality, is also worse than in Model 1. It should be mentioned that the visibility variable has a high P-value of 0.903.

The variance inflation factors (VIF) are now calculated below. **Model 3** takes into account the

high P-value of visibility and the VIF values.

[38]:
```
vals = [VIF(X_2, i)
    for i in range(1, X_2.shape[1])]
vif = pd.DataFrame({'vif':vals},
                   index=X_2.columns[1:])
print(vif)
```

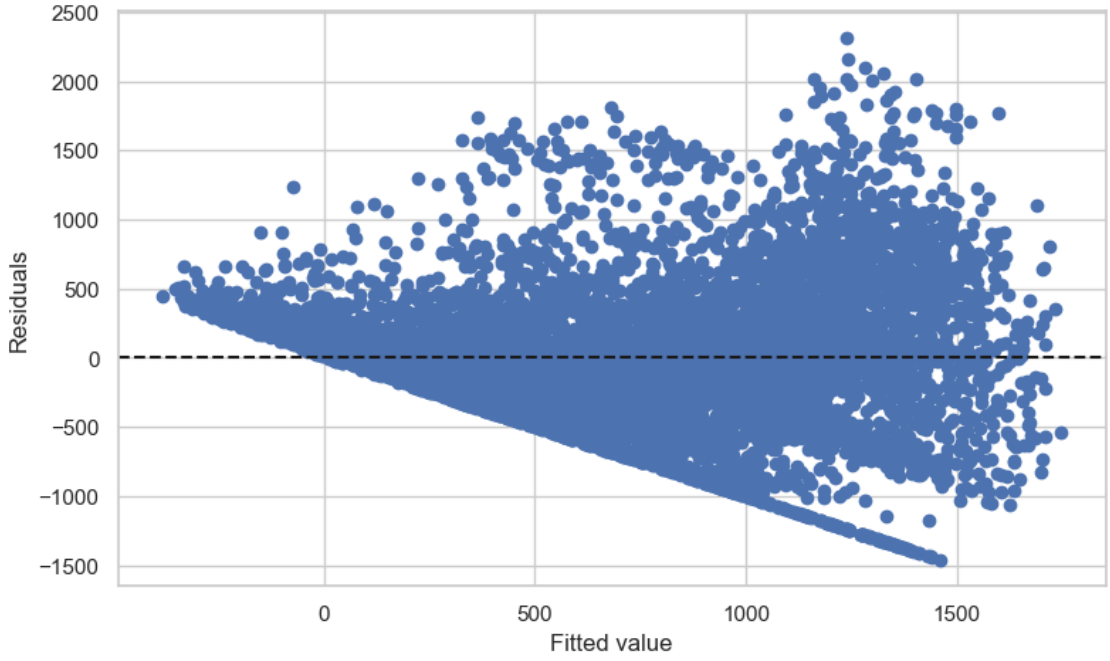
```
                            vif
hour                   1.114018
temperature           86.445397
humidity              19.585110
visibility             1.621341
dew_point_temperature 113.512721
solar_radiation        1.896910
seasons                1.760135
year                   1.253176
```

[39]:
```
# MODEL 3 (5 significant variables)
data_3 = df.columns.drop(['rented_bike_count', 'date', 'wind_speed',␣
 ↪'rainfall', 'snowfall', 'holiday', 'day', 'month', 'weekday',
                           'visibility', 'dew_point_temperature', 'year'])
X_3 = MS(data_3).fit_transform(df)
y_3 = pd.DataFrame({'rented_bike_count': df['rented_bike_count']})
model_3 = sm.OLS(y_3, X_3)
results_3 = model_3.fit()
# results_3.summary(model_3)
print('Results model 3 (5 significant variables)')
print('R-squared:', results_3.rsquared.round(4))
print('Adj. R-squared:', results_3.rsquared_adj.round(4))
print('F-statistic:', results_3.fvalue.round(4))
print('AIC:', results_3.aic.round(4))
print('---------------------------------')
print('Values for all variables:')
print(results_3.pvalues.round(4))
```

```
Results model 3 (5 significant variables)
R-squared: 0.4686
Adj. R-squared: 0.4683
F-statistic: 1544.026
AIC: 132673.2223
---------------------------------
Values for all variables:
intercept          0.0
hour               0.0
temperature        0.0
humidity           0.0
solar_radiation    0.0
```

```
seasons              0.0
dtype: float64
```

[40]:
```
# Visualization of the residuals against the estimated values (fitted values)␣
 ↪of a regression analysis
plot_3 = subplots(figsize=(8,5))[1]
plot_3.scatter(results_3.fittedvalues, results_3.resid)  # Creates the␣
 ↪scatterplot
plot_3.set_xlabel('Fitted value')
plot_3.set_ylabel('Residuals')
plot_3.axhline(0, c='k', ls='--');
```



Evaluation of the results:
Model 3 has the same high model performance as Model 2 (R^2 = 0.469, Adj. R^2 = 0.468).
The residual plot shows a clear funnel shape, which indicates that the variance of the residuals is not constant. The systematic deviation from the zero line indicates that further improvements to the model are required.
The F-statistic indicates that Model 3 (F-statistic = 1544) is significantly better than Model 2 (F-statistic = 966). The higher F-statistic indicates that the additional variables in Model 3 may have a stronger influence on model performance.
However, Model 3 has a lower AIC value (AIC = 132673) than Model 2 (AIC = 132672) and Model 1 (AIC = 132387), which indicates a lower model quality.
Since Model 3 reduces significantly the model complexity, we continue to use Model 3 in the following steps and accept a lower model performance.

2. Using rented_bike_count_intervalls for the multiple linear regression model

During the modeling phase, the idea arose to predict an interval for rented_bike_count instead of the numerical target variable (exact number of rented_bike_count).
This could result in improved model performance , as the predictions are less precise.
In the following, the exact number of the variable rented_bike_count is therefore rounded to rented_bike_count_interval, whereby intervals are generated in steps of 50 (e.g. 257 -> 250).
Model 4 is then trained with the significant variables from Model 3 to predict rented_bike_count_interval.

[41]:
```
def round_to_nearest_50(x):
    return round(x / 50) * 50
```

[42]:
```
df_intervall = df.copy()
df_intervall['rented_bike_count_intervall'] = df_intervall['rented_bike_count'].
 ↪apply(round_to_nearest_50)
df_intervall = df_intervall.drop(columns = ['rented_bike_count'])
df_intervall.head()
```

[42]:

	date	hour	temperature	humidity	wind_speed	visibility	dew_point_temperature	solar_radiation	rainfall	snowfall	seasons	holiday	day	month	year	weekday	rented_bike_count_intervall
0	2017-12-01	0	-5.2	37	2.2	2000	-17.6	0.0	0.0	0.0	0	0	1	12	2017	5	250
1	2017-12-01	1	-5.5	38	0.8	2000	-17.6	0.0	0.0	0.0	0	0	1	12	2017	5	200
2	2017-12-01	2	-6.0	39	1.0	2000	-17.7	0.0	0.0	0.0	0	0	1	12	2017	5	150
3	2017-12-01	3	-6.2	40	0.9	2000	-17.6	0.0	0.0	0.0	0	0	1	12	2017	5	100
4	2017-12-01	4	-6.0	36	2.3	2000	-18.6	0.0	0.0	0.0	0	0	1	12	2017	5	100

[43]:
```
# MODEL 4 (5 variables with rented_bike_count_intervall as target)
data_4 = df.columns.drop(['rented_bike_count', 'date', 'wind_speed',␣
 ↪'rainfall', 'snowfall', 'holiday', 'day', 'month', 'weekday',
                          'visibility', 'dew_point_temperature', 'year'])
X_4 = MS(data_3).fit_transform(df)
y_4 = pd.DataFrame({'rented_bike_count_intervall':␣
 ↪df_intervall['rented_bike_count_intervall']})
model_4 = sm.OLS(y_4, X_4)
results_4 = model_4.fit()
```

```
# results_3.summary(model_3)
print('Results model 4 (5 variables with rented_bike_count_intervall as␣
 ↪target)')
print('R-squared:', results_4.rsquared.round(4))
print('Adj. R-squared:', results_4.rsquared_adj.round(4))
print('F-statistic:', results_4.fvalue.round(4))
print('AIC:', results_4.aic.round(4))
print('---------------------------------')
print('Values for all variables:')
print(results_4.pvalues.round(4))
```

```
Results model 4 (5 variables with rented_bike_count_intervall as target)
R-squared: 0.468
Adj. R-squared: 0.4677
F-statistic: 1540.1809
AIC: 132692.7723
---------------------------------
Values for all variables:
intercept          0.0
hour               0.0
temperature        0.0
humidity           0.0
solar_radiation    0.0
seasons            0.0
dtype: float64
```

Evaluation of the results:
The results of Model 3 and Model 4 differ only minimally. However, Model 4 has a lower model performance. In addition, Model 3 is based on the continuous target variable rented_bike_count, which enables more precise predictions. Model 4 is therefore no longer used.

3. Optimization of the linear regression model with interaction terms

In order to further increase the performance of Model 3, linear interaction terms are to be included in the model. These make it possible to take into account the interactions between the independent variables and thus better depict potentially hidden relationships. The aim is to increase the explanatory power of the model.

The following four linear interaction terms were defined for this purpose: - hour * temperature - temperature * seasons - hour * solar_radiation - seasons * humidity

[44]:
```
# MODEL 5 (5 variables with 4 linear interaction terms included)
data_5 = ['hour', 'temperature', 'humidity', 'solar_radiation', 'seasons',
          ('hour','temperature'), ('temperature','seasons'),␣
 ↪('hour','solar_radiation'), ('seasons','humidity')]
X_5 = MS(data_5).fit_transform(df)
y_5 = pd.DataFrame({'rented_bike_count': df['rented_bike_count']})
model_5 = sm.OLS(y_5, X_5)
results_5 = model_5.fit()
```

```
# results_5.summary(model_5)
print('Results model 5 (5 variables with 4 linear interaction terms included)')
print('R-squared:', results_5.rsquared.round(4))
print('Adj. R-squared:', results_5.rsquared_adj.round(4))
print('F-statistic:', results_5.fvalue.round(4))
print('AIC:', results_5.aic.round(4))
print('----------------------------------')
print('Values for all variables:')
print(results_5.pvalues.round(4))
```

```
Results model 5 (5 variables with 4 linear interaction terms included)
R-squared: 0.5218
Adj. R-squared: 0.5213
F-statistic: 1060.9634
AIC: 131757.0933
----------------------------------
Values for all variables:
intercept               0.0000
hour                    0.0000
temperature             0.0000
humidity                0.0000
solar_radiation         0.0000
seasons                 0.0000
hour:temperature        0.0000
temperature:seasons     0.0000
hour:solar_radiation    0.0548
seasons:humidity        0.0000
dtype: float64
```

Evaluation of the results:
Model 5 shows a clear improvement in model quality compared to Model 3. With an R^2 value of 0.5218 and an adjusted R^2 of 0.5213, it explains more variance in the target variables than Model 3 (R^2: 0.4686, Adj. R^2: 0.4683).
The F-statistic of Model 5 (1060.96) is also significantly higher, which indicates a stronger significance of the variables and interaction terms, while the AIC value (131757.09) is considerably lower and thus indicates a better fit of the model.
Overall, Model 5 offers more precise modeling by taking the linear interaction terms into account and outperforms Model 3 in all important key figures.

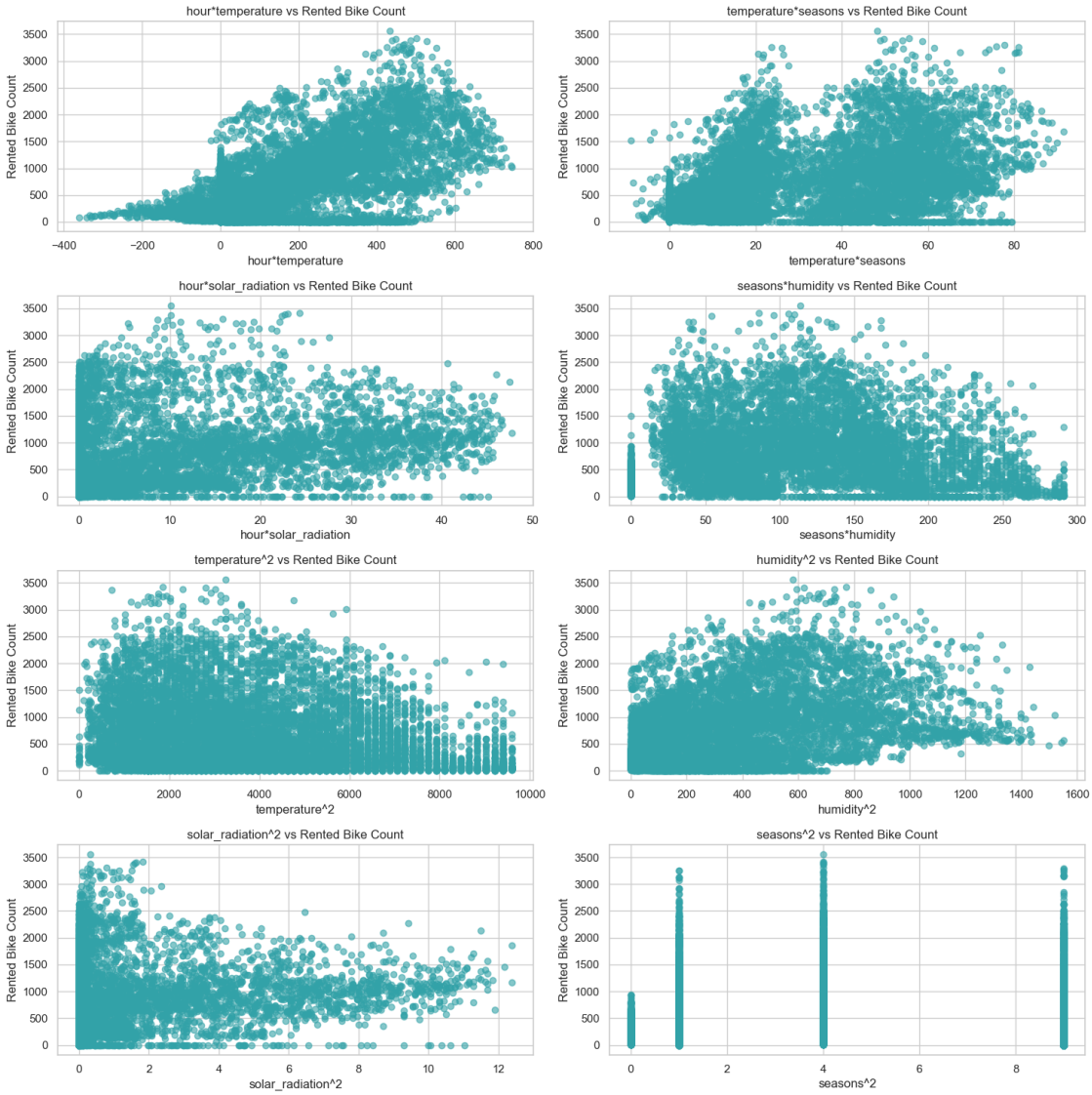
[45]:
```
# Model 6 (5 variables with 4 linear interaction terms and 4 polynomial␣
 ↪interactions terms included)
# Define the data and the X_6 and y_6 variables
data_6 = ['hour', 'temperature', 'humidity', 'solar_radiation', 'seasons']
X_6 = df[data_6].copy()
y_6 = df['rented_bike_count']

# Manual creation of the polynomial interaction terms
X_6['hour*temperature'] = (X_6['hour'] * X_6['temperature'])
```

```
X_6['temperature*seasons'] = (X_6['temperature'] * X_6['seasons'])
X_6['hour*solar_radiation'] = (X_6['hour'] * X_6['solar_radiation'])
X_6['seasons*humidity'] = (X_6['seasons'] * X_6['humidity'])
X_6['temperature^2'] = X_6['humidity'] ** 2
X_6['humidity^2'] = X_6['temperature'] ** 2
X_6['solar_radiation^2'] = X_6['solar_radiation'] ** 2
X_6['seasons^2'] = X_6['seasons'] ** 2

# Selection of features
selected_poly_features = ['hour', 'temperature', 'humidity', 'solar_radiation',␣
 ↪'seasons',
                          'hour*temperature', 'temperature*seasons',␣
 ↪'hour*solar_radiation', 'seasons*humidity',
                          'temperature^2', 'humidity^2', 'solar_radiation^2',␣
 ↪'seasons^2']
X_6_poly = X_6[selected_poly_features]

# Train the model
X_6_poly = sm.add_constant(X_6_poly)
model_6 = sm.OLS(y_6, X_6_poly)
results_6 = model_6.fit()
#
print('Results model 6 (5 variables with 4 linear interaction terms and 4␣
 ↪polynomial interactions terms included)')
print('R-squared:', results_6.rsquared.round(4))
print('Adj. R-squared:', results_6.rsquared_adj.round(4))
print('F-statistic:', results_6.fvalue.round(4))
print('AIC:', results_6.aic.round(4))
print('---------------------------------')
print('Values for all variables:')
print(results_6.pvalues.round(4))
```

```
Results model 6 (5 variables with 4 linear interaction terms and 4 polynomial
interactions terms included)
R-squared: 0.5464
Adj. R-squared: 0.5457
F-statistic: 810.2481
AIC: 131303.8187
---------------------------------
Values for all variables:
const                  0.5351
hour                   0.0000
temperature            0.0000
humidity               0.0000
solar_radiation        0.0185
seasons                0.0000
hour*temperature       0.0000
```

```
temperature*seasons     0.2306
hour*solar_radiation    0.0000
seasons*humidity        0.0000
temperature^2           0.0000
humidity^2              0.0000
solar_radiation^2       0.0000
seasons^2               0.0000
dtype: float64
```

[46]:
```python
plt.figure(figsize=(15, 15))

for idx, feature in enumerate(X_6.columns[5:], start=1):
    plt.subplot(4, 2, idx)
    plt.scatter(X_6[feature], y_6, alpha=0.6, color='#32a2a8')
    plt.title(f'{feature} vs Rented Bike Count')
    plt.xlabel(feature)
    plt.ylabel('Rented Bike Count')
    plt.grid(True)

plt.tight_layout()
plt.show()
```



Evaluation of the results:
Model 6 offers the best explanatory power (highest R^2) and the best model quality (lowest AIC) among the three models, making it an excellent choice for more precise predictions. However, the slightly lower F-statistic and the non-significant term temperature*seasons could indicate a possible overfitting.
Compared to Model 3 and Model 5, Model 6 shows significant improvements, especially in terms of model performance and goodness of fit.

Advantages of Model 6: - Highest explanatory power (highest R^2 and adjusted R^2). - Best model quality (lowest AIC). - Takes into account non-linear and complex interactions, which improves the accuracy of the predictions.

Disadvantages of Model 6: - Higher model complexity, which makes interpretability more difficult. - Slightly reduced significance compared to Model 5 (lower F-statistic). - Not all terms are significant,

indicating possible overfitting.

In the following section, Model 6 is checked for overfitting using the testing-training approach.

4. Training / Testing of model 6 (polynomial interactions terms included)

Model 6 has the highest model performance, but may be prone to overfitting. To check the generalization capability of Model 6 and identify potential overfitting, the model is evaluated using a testing-training approach. The data is split into a training data set and a test data set to ensure that the model not only describes the training data well, but also provides robust predictions on unknown data.

[47]:
```
# Splitting data_6 into training-dataset and test-dataset
X_6_test_train = X_6_poly.copy()
y_6_test_train = y_6.copy()
X_train, X_test, y_train, y_test = train_test_split(X_6_test_train,␣
 ↪y_6_test_train, test_size = 0.30, random_state = 1)
```

[48]:
```
# Fitting training-dataset to model
model_6_test_train = sm.OLS(y_6_test_train, X_6_test_train)
results_6_test_train = LinearRegression().fit(X_train, y_train)
```

[49]:
```
# Prediction of model
y_pred_test_train = results_6_test_train.predict(X_test)
```

[50]:
```
# Training accuracy of model
results_6_test_train.score(X_train, y_train)
```

[50]:
```
0.5503933298232939
```

[51]:
```
# Test accuracy of model
results_6_test_train.score(X_test, y_test)
```

[51]:
```
0.535657327618652
```

Evaluation of the Results:
Overall, the model shows good generalization ability, as the R^2 values for training (0.550) and testing (0.536) are relatively close to each other. This indicates that there is no significant overfitting and that the model provides robust predictions even on unknown data.

5. Compute predicitions for rented_bike_count based on model 5 and model 6

We now want to predict the target variable rented_bike_count for three possible examples. These are the values for the three examples: - hour: 8, 12, 16 - temperature: 25.5, 24.0, 27.2 - humidity: 55, 70, 85 - solar_radiation: 0.7, 0.5, 0.8 - seasons: 2, 3, 3

[52]:
```
# Prediciton for model 5 (5 variables with 4 linear interaction terms included)
# Create new data to be predicted for model 5
new_df_5 = pd.DataFrame({
    'hour': [8, 12, 16],
```

```
    'temperature': [25.5, 24.0, 27.2],
    'humidity': [55, 70, 85],
    'solar_radiation': [0.7, 0.5, 0.8],
    'seasons': [2, 3, 3],
})

# Create predictions for model 5
new_prediction_df_5 = MS(data_5).fit_transform(new_df_5)
new_predictions_5 = results_5.get_prediction(new_prediction_df_5)
predicted_mean_5 = new_predictions.predicted_mean
print('Predictions model 5:', predicted_mean_5)
```

```
Predictions model 5: [359.03361294 504.43910789  68.22262305]
```

[53]:
```
# Predictions and confidence intervals for Model 5
conf_int_5 = pd.DataFrame(new_predictions_5.conf_int(alpha=0.05),␣
 ↪columns=['confidence_low_5', 'confidence_high_5'])
predictions_df_5 = pd.DataFrame({
    'confidence_low_5': conf_int_5['confidence_low_5'].values,
    'predicted_mean_5': new_predictions_5.predicted_mean,
    'confidence_high_5': conf_int_5['confidence_high_5'].values
})
predictions_df_5
```

[53]:

	confidence_low_5	predicted_mean_5	confidence_high_5
0	865.971697	890.160269	914.348841
1	881.587851	906.664158	931.740465
2	949.788865	989.282808	1028.776750

[54]:
```
# Prediction for model 6 (5 variables with 4 linear interaction terms and 4␣
 ↪polynomial interactions terms included)
# Create new data to be predicted for model 6
new_df_6 = pd.DataFrame({
    'hour': [8, 12, 16],
    'temperature': [25.5, 24.0, 27.2],
    'humidity': [55, 70, 85],
    'solar_radiation': [0.7, 0.5, 0.2],
    'seasons': [2, 3, 2],
})

# Calculate the interaction terms and polynomial terms for model 6
new_df_6['hour*temperature'] = new_df_6['hour'] * new_df_6['temperature']
new_df_6['temperature*seasons'] = new_df_6['temperature'] * new_df_6['seasons']
new_df_6['hour*solar_radiation'] = new_df_6['hour'] *␣
 ↪new_df_6['solar_radiation']
new_df_6['seasons*humidity'] = new_df_6['seasons'] * new_df_6['humidity']
new_df_6['temperature^2'] = new_df_6['temperature'] ** 2
```

```
new_df_6['humidity^2'] = new_df_6['humidity'] ** 2
new_df_6['solar_radiation^2'] = new_df_6['solar_radiation'] ** 2
new_df_6['seasons^2'] = new_df_6['seasons'] ** 2

# Add constant for model 6
new_prediction_df_6 = sm.add_constant(new_df_6)

# Create predictions for Model 6
new_predictions_6 = results_6.get_prediction(new_prediction_df_6)
predicted_mean_6 = new_predictions_6.predicted_mean
print('Predictions model 6:', predicted_mean_6)
```

```
Predictions model 6: [ 561.52978627  196.96127091 -129.36653361]
```

[55]:
```
# Predictions and confidence intervals for Model 6
conf_int_6 = pd.DataFrame(new_predictions_6.conf_int(alpha=0.05),␣
 ↪columns=['confidence_low_6', 'confidence_high_6'])
predictions_df_6 = pd.DataFrame({
    'confidence_low_6': conf_int_6['confidence_low_6'].values,
    'predicted_mean_6': new_predictions_6.predicted_mean,
    'confidence_high_6': conf_int_6['confidence_high_6'].values
})
predictions_df_6
```

[55]:

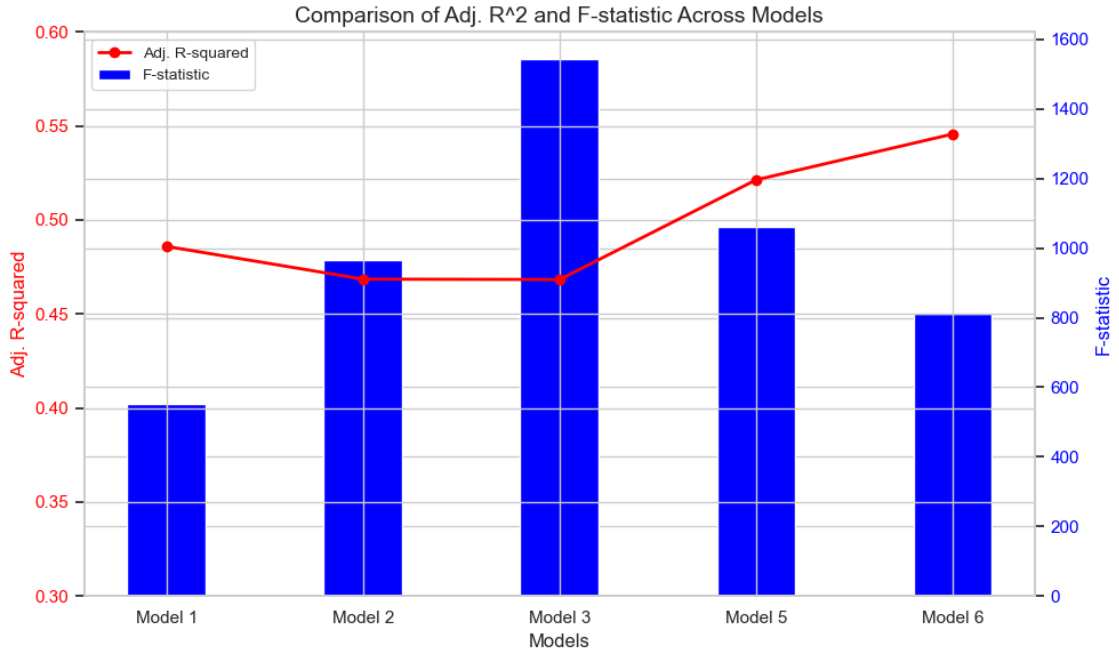
	confidence_low_6	predicted_mean_6	confidence_high_6
0	324.853215	561.529786	798.206357
1	-218.572706	196.961271	612.495248
2	-766.434683	-129.366534	507.701616

6. Visualizations to compare the models

In the following, we would like to compare the results of adj R^2 and F-statistic in a visualization.

[56]:
```
model_comparison = pd.DataFrame({
    'Model': ['Model 1', 'Model 2', 'Model 3', 'Model 5', 'Model 6'],
    'Adj_R_squared': [results_1.rsquared_adj, results_2.rsquared_adj, results_3.
 ↪rsquared_adj,
                      results_5.rsquared_adj, results_6.rsquared_adj],
    'F_statistic': [results_1.fvalue, results_2.fvalue, results_3.fvalue,
                    results_5.fvalue, results_6.fvalue]
})

# Create plot
fig, ax1 = plt.subplots(figsize=(10, 6))
x = np.arange(len(model_comparison['Model']))

# Create bar chart: F-statistic on the right axis
ax2 = ax1.twinx()
```

```
ax2.bar(x, model_comparison['F_statistic'], color='blue', alpha=1, width=0.4,␣
 ↪label='F-statistic')
ax2.set_ylabel('F-statistic', fontsize=12, color='blue')
ax2.tick_params(axis='y', labelcolor='blue')

# Create line chart: Adj. R^2 on the left axis
ax1.plot(x, model_comparison['Adj_R_squared'], label='Adj. R-squared',␣
 ↪color='red', marker='o', linewidth=2, zorder=3)
ax1.set_xlabel('Models', fontsize=12)
ax1.set_ylabel('Adj. R-squared', fontsize=12, color='red')
ax1.set_ylim(0.3, 0.6)
ax1.tick_params(axis='y', labelcolor='red')
ax1.set_zorder(2)
ax1.patch.set_visible(False)

# Define labels and titles
ax1.set_xticks(x)
ax1.set_xticklabels(model_comparison['Model'])
ax1.set_title('Comparison of Adj. R^2 and F-statistic Across Models',␣
 ↪fontsize=14)
lines_1, labels_1 = ax1.get_legend_handles_labels()
lines_2, labels_2 = ax2.get_legend_handles_labels()
ax1.legend(lines_1 + lines_2, labels_1 + labels_2, loc='upper left',␣
 ↪fontsize=10)

# Plot the final visualization
plt.tight_layout()
plt.show()
```



Now let's compare the R^2 for training and testing for the different models in the next visualization.

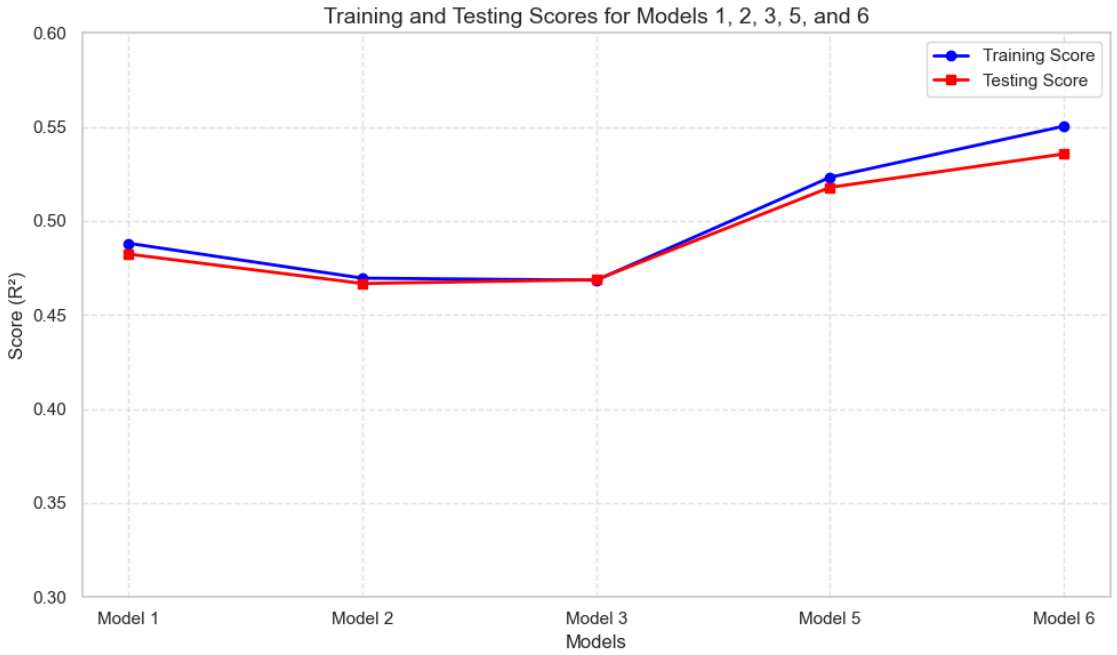
[57]:
```
# Function for calculating the training and testing scores
def calculate_model_scores(X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=1)
    model = LinearRegression()
    model.fit(X_train, y_train)
    train_score = model.score(X_train, y_train)
    test_score = model.score(X_test, y_test)
    return train_score, test_score

# Prepare dataset for models and calculate training/testing scores
models_data = {
    "Model 1": (X_1, y_1),
    "Model 2": (X_2, y_2),
    "Model 3": (X_3, y_3),
    "Model 5": (X_5, y_5),
    "Model 6": (X_6_poly, y_6),
}
train_test_scores = {}
for model_name, (X, y) in models_data.items():
    train_score, test_score = calculate_model_scores(X, y)
    train_test_scores[model_name] = (train_score, test_score)

# Save results in DataFrame
scores_df = pd.DataFrame(
    train_test_scores, index=["Training Score", "Testing Score"]
).T
scores_df
```

[57]:

	Training Score	Testing Score
Model 1	0.488123	0.482366
Model 2	0.469577	0.466746
Model 3	0.468510	0.468746
Model 5	0.523215	0.517884
Model 6	0.550393	0.535657

[58]:
```
# Plot of the training and testing scores
plt.figure(figsize=(10, 6))
plt.plot(scores_df.index, scores_df["Training Score"], marker='o',␣
 ↪label="Training Score", color="blue", linewidth=2)
plt.plot(scores_df.index, scores_df["Testing Score"], marker='s',␣
 ↪label="Testing Score", color="red", linewidth=2)
```

```
# Define labels and titles
plt.xlabel("Models", fontsize=12)
plt.ylabel("Score (R²)", fontsize=12)
plt.title("Training and Testing Scores for Models 1, 2, 3, 5, and 6",␣
 ↪fontsize=14)
plt.ylim(0.3, 0.6)  # Begrenzung der y-Achse
plt.xticks(rotation=0)
plt.legend()
plt.grid(True, linestyle="--", alpha=0.6)

# Plot the final visualization
plt.tight_layout()
plt.show()
```



1.13 K nearest neighbors and cross validation

In this part, we are going to try a new model the K Nearest Neighbors with the same features as we have chosen before. We are using the GridSearch to know the best neighbors numbers. Afterward, we will be able to split the dataset in train et test set, train the model, and do some predictions. We will have the score f the regression.

[59]:
```
#Data collection
X = pd.DataFrame({'temperature': df['temperature'], 'humidity' :␣
 ↪df['humidity'], 'hour' : df['hour'], 'visibility' : df['visibility'],␣
 ↪'dew_point_temperature' : df['dew_point_temperature'], 'solar_radiation' :␣
 ↪df['solar_radiation'], 'seasons':df['seasons'],'year' : df['year'] })
```

The type date is compatible with KNN

[60]:
```
y = pd.DataFrame({ 'rented_bike_count': df['rented_bike_count']})
y.head()
```

[60]:
```
   rented_bike_count
0                254
1                204
2                173
3                107
4                 78
```

[61]:
```
X_train, X_val, y_train, y_val = train_test_split(X,y, random_state =7,␣
↪train_size=0.75)
```

[62]:
```
knn = KNeighborsRegressor(n_neighbors=7)
```

[63]:
```
from sklearn.model_selection import GridSearchCV
```

[64]:
```
#cross validation to find the best hyperparmeters (numbers of neighbors)
parameters = {"n_neighbors":[3,7,9,12,15]}
clf = GridSearchCV(knn, parameters,cv=5)
clf.fit(X_train, y_train)

knn_model = clf.best_estimator_
print (knn_model,'\n')
knn_model.fit(X_train,y_train)
```

```
KNeighborsRegressor(n_neighbors=7)
```

[64]:
```
KNeighborsRegressor(n_neighbors=7)
```

The GridSearchCV function is a cross validation method that find the best perameters corresponding to model used. In this case it's number neigbor that the gridSearchCV will optimize

[65]:
```
#Gets the scrores
y_pred_train = knn_model.predict(X_train)
y_pred_test = knn_model.predict(X_val)
r2_train = r2_score(y_train, y_pred_train)
r2_test = r2_score(y_val, y_pred_test)
print ("The real training model accuracy is: \n r2_train = ",r2_train)
print ("The real testing model accuracy is: \n r2_test = ",r2_test)
```

```
The real training model accuracy is:
 r2_train =  0.6625198174421785
The real testing model accuracy is:
 r2_test =  0.5544623968856592
```

We can see that we have a small overfitting with a training score of 0.66 and a testing score of 0.55.

After, we want to check manually if the number of neighbors is really teh best one with a for loop :

[66]:
```
# loop to check if the GridSchearchCV have found the best number of neighbor
m = 0
l_test = []
l_train = []
K= []
n = 0
for k in range (1,10):
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(X_train,y_train)
    y_pred_test = knn.predict(X_val)
    y_pred_train = knn.predict(X_train)
    r2_train = r2_score(y_train, y_pred_train)
    r2_test = r2_score(y_val, y_pred_test)
    l_test = l_test +[r2_test]
    l_train = l_train +[r2_train]
    K=K+[k]
    if r2_test>=m :
        m=r2_test
        n=k
[m,n]
```

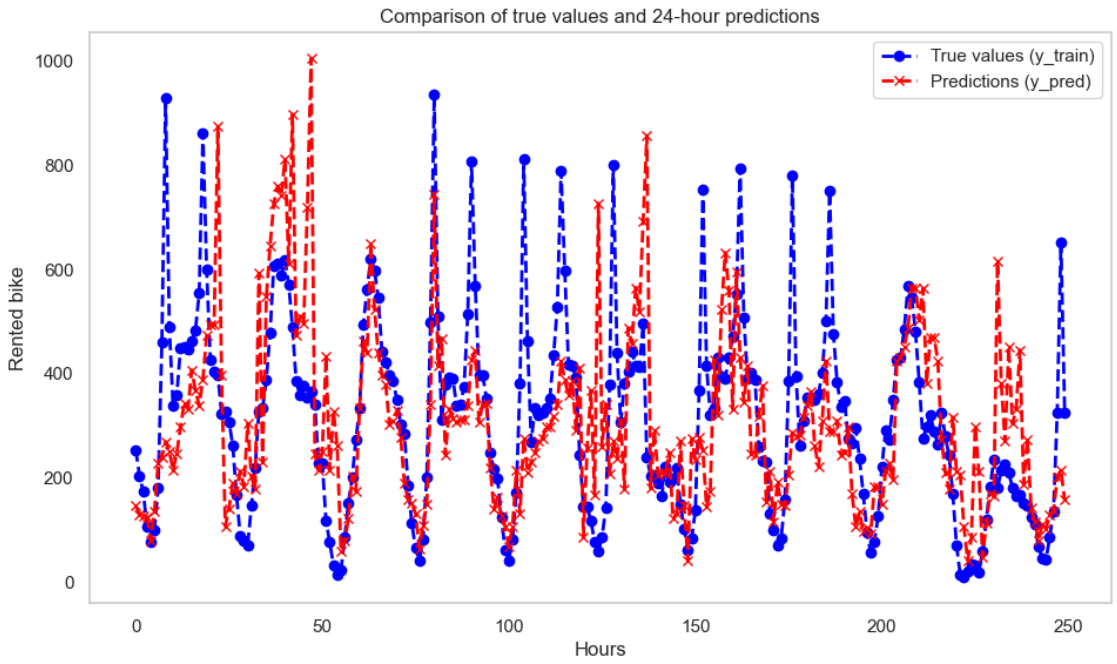
[66]:
```
[0.5544623968856592, 7]
```

[67]:
```
#Plot the figure that the corresponding testing and training score compared to␣
 ↪the number of neighbors
plt.figure()
plt.plot(K, l_test, 'o', color='blue',linestyle='dashed',label="testing Score",
     linewidth=2)
plt.plot(K, l_train, 'o', color='red',linestyle='dashed',label="Training Score",
     linewidth=2)
plt.title('Comparison of number neighbors of n R2 scores  ')
plt.xlabel('number neighbors')
plt.ylabel('R2 scores')
plt.legend()
plt.grid()
plt.show()
```

Indeed, with this method, we can see that the best number of neighbors is 7, as we found with GridSearchCV. To conclude this part, we can say that the GridSearchCV function is efficient in finding the optimal model parameters. However, we can also observe that the scores for this regression are low.

[68]:
```
#We plot the training set and its predictions
y_pred_train_1 = knn_model.predict(X)

plt.figure(figsize=(10, 6))
plt.plot(y[0:250], 'o', label='True values (y_train)',␣
 ↪color='blue',linestyle='dashed',
    linewidth=2)
plt.plot(y_pred_train_1[0:250], 'x', label='Predictions (y_pred)',␣
 ↪color='red',linestyle='dashed',
    linewidth=2)

plt.title('Comparison of true values and 24-hour predictions')
plt.xlabel('Hours')
plt.ylabel('Rented bike')
plt.legend()
plt.grid()
```

```
plt.show()
```



This graph clearly shows that the model is not efficient enough, even with the training set. The predictions follow the daily trend, with more bikes rented during the day than at night. However, we can clearly see significant discrepancies between the predictions and the true values. Indeed, the K-Nearest Neighbor algorithm is not the most efficient model for forecasting the number of rented bikes.

1.14 Random Forest Regressor

In this part we are going to train and test the random forest model in order to evaluate if this model is fitting better than the other already trained and implemented.

[69]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler, OneHotEncoder

rfr = RandomForestRegressor(n_estimators=100, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Train the model
rfr.fit(X_train, y_train)

# Predict on the test set
```

```
y_pred = rfr.predict(X_test)

# Evaluate the model
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Random Forest Regressor Performance:")
print(f"Root Mean Square Error (RMSE): {rmse:.2f}")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"R² Score: {r2:.2f}")
```

```
C:\Users\Raphael Preis\anaconda3\Lib\site-packages\sklearn\base.py:1474:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples,), for example using
ravel().
  return fit_method(estimator, *args, **kwargs)

Random Forest Regressor Performance:
Root Mean Square Error (RMSE): 312.10
Mean Absolute Error (MAE): 184.17
R² Score: 0.77
```

As first we find out an overall better R^2 score.

Next step is implementing random forest regressor with 5 estimators [100, 200, 300, 400, 500], 3 minimum samples split with value [2, 5, 10] and 3 minimum samples leaf of value [1, 2, 4].

[70]:
```
from sklearn.model_selection import RandomizedSearchCV
param_grid = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None]  # Ensure 'auto' is removed
}

random_search = RandomizedSearchCV(
    estimator=RandomForestRegressor(random_state=42),
    param_distributions=param_grid,
    n_iter=50,
    scoring='neg_mean_squared_error',
    cv=3,
    verbose=2,
    random_state=42,
    n_jobs=-1
)

random_search.fit(X_train, y_train)
```

```
best_params = random_search.best_params_
best_model = random_search.best_estimator_

y_pred_best = best_model.predict(X_test)
rmse_best = np.sqrt(mean_squared_error(y_test, y_pred_best))
mae_best = mean_absolute_error(y_test, y_pred_best)
r2_best = r2_score(y_test, y_pred_best)

print("Best Hyperparameters:", best_params)
print(f"Tuned Random Forest Regressor Performance:")
print(f"Root Mean Square Error (RMSE): {rmse_best:.2f}")
print(f"Mean Absolute Error (MAE): {mae_best:.2f}")
print(f"R² Score: {r2_best:.2f}")
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits

C:\Users\Raphael Preis\anaconda3\Lib\site-packages\sklearn\base.py:1474:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples,), for example using
ravel().
  return fit_method(estimator, *args, **kwargs)

Best Hyperparameters: {'n_estimators': 500, 'min_samples_split': 2,
'min_samples_leaf': 2, 'max_features': 'log2', 'max_depth': None}
Tuned Random Forest Regressor Performance:
Root Mean Square Error (RMSE): 309.21
Mean Absolute Error (MAE): 183.14
R² Score: 0.77
```

2 Cross Validation code for Random Forest Regression

In order to try to find a better solution compared to the straight implementation of Random Forest model cross validation is implemented performing a 5 fold cross validation.

[71]:
```
import warnings
warnings.filterwarnings('ignore')
```

[72]:
```
from sklearn.metrics import make_scorer

# Define the Random Forest Regressor
model = RandomForestRegressor(n_estimators=200, random_state=42)

# Define a custom scoring function for RMSE
scorer = make_scorer(mean_squared_error, greater_is_better=False)

# Perform 5-fold cross-validation
cv_scores = cross_val_score(
```

```
    estimator=model,
    X=X_train,
    y=y_train,
    cv=5,  # Number of folds
    scoring=scorer,
    n_jobs=-1  # Use all processors
)

# Convert negative MSE to RMSE for interpretability
rmse_scores = np.sqrt(-cv_scores)

# Output cross-validation results
print(f"Cross-Validation RMSE Scores: {rmse_scores}")
print(f"Mean RMSE: {np.mean(rmse_scores):.2f}")
print(f"Standard Deviation of RMSE: {np.std(rmse_scores):.2f}")
```

```
Cross-Validation RMSE Scores: [312.65925475 306.4255431  323.56634247
299.0007597  326.5604492 ]
Mean RMSE: 313.64
Standard Deviation of RMSE: 10.32
```

[73]:
```
from sklearn.ensemble import GradientBoostingRegressor

# Initialize the Gradient Boosting Regressor
gbr = GradientBoostingRegressor(
    n_estimators=200,
    learning_rate=0.1,
    max_depth=3,
    subsample=0.8,
    random_state=42
)

# Fit the model to the training data
gbr.fit(X_train, y_train)

# Predict on test data
y_pred = gbr.predict(X_test)
```

[74]:
```
# Make predictions on the test set
y_pred = best_model.predict(X_test)

# Calculate evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

# Print evaluation results
```

```
print("Performance of Tuned Gradient Boosting Regressor:")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R² Score: {r2:.2f}")
```

```
Performance of Tuned Gradient Boosting Regressor:
Mean Absolute Error (MAE): 183.14
Root Mean Squared Error (RMSE): 309.21
R² Score: 0.77
```

After all we find the same R^2 score of 0.77 and a small and negligible improvement for RMSE and MAE.

In the end we conclude that an R^2 of 0.77 is the best value we can achieve even if compared with the training R^2 = 0.93 we can observe a non negligible problem of overfitting of our model.

[75]:
```
# Evaluate R² on training data
train_r2 = best_model.score(X_train, y_train)

# Evaluate R² on testing data
test_r2 = best_model.score(X_test, y_test)

# Print results
print(f"Training R²: {train_r2:.2f}")
print(f"Testing R²: {test_r2:.2f}")
```

```
Training R²: 0.93
Testing R²: 0.77
```

In addition to that and trying to see if the overfitting problem can be solved Boostrapping method is implemented with the Gradient boosting regressor function and a number of bootstraps equal to 100.

[76]:
```
from sklearn.utils import resample
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

n_bootstrap = 100

models = []

for i in range(n_bootstrap):
    X_resampled, y_resampled = resample(X_train, y_train, random_state=42 + i)

    model = GradientBoostingRegressor(random_state=42)
    model.fit(X_resampled, y_resampled)

    models.append(model)
```

```
y_pred_bootstrap = np.mean([model.predict(X_test) for model in models], axis=0)

# Evaluate performance
mae = mean_absolute_error(y_test, y_pred_bootstrap)
rmse = np.sqrt(mean_squared_error(y_test, y_pred_bootstrap))
r2 = r2_score(y_test, y_pred_bootstrap)

# Print results
print("Bootstrapped Gradient Boosting Regressor Performance:")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R² Score: {r2:.2f}")
```

```
Bootstrapped Gradient Boosting Regressor Performance:
Mean Absolute Error (MAE): 199.93
Root Mean Squared Error (RMSE): 325.73
R² Score: 0.75
```

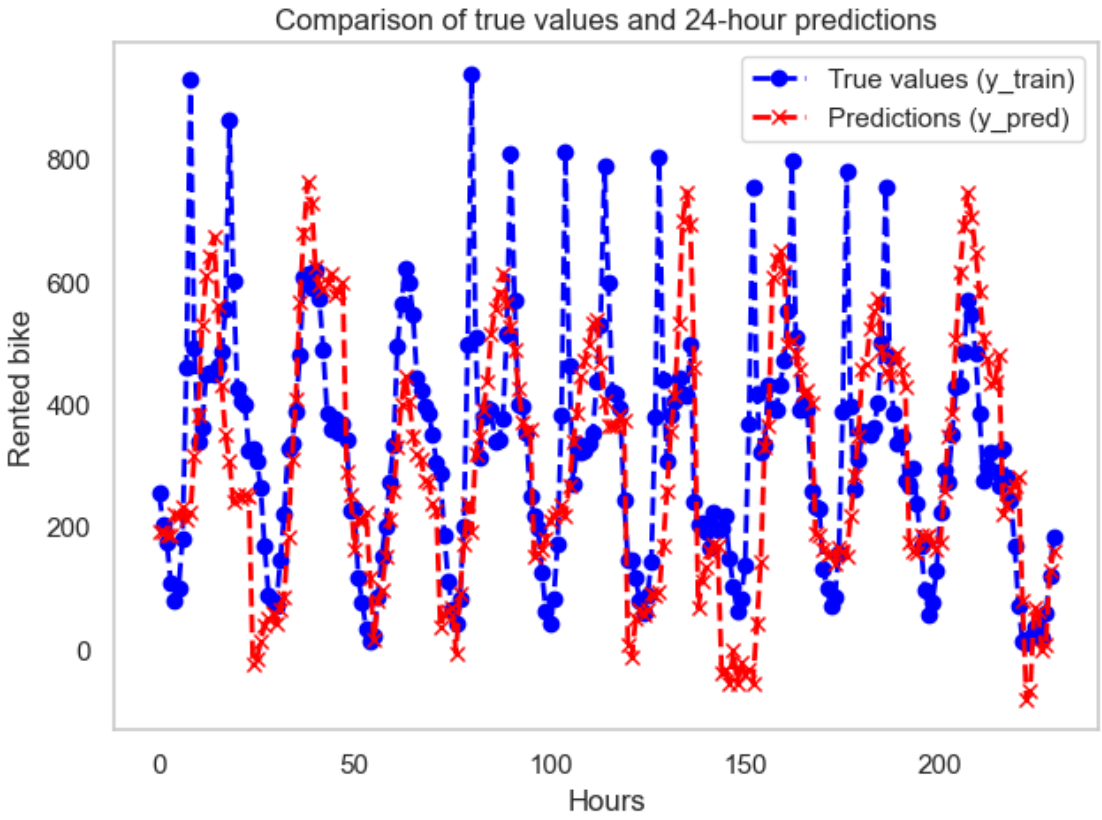
The outcome is not better than before: R^2 equal to 0.75 and worse values for MAE and RMSE. This leaves us to conclude that for the moment the best model implemented so far is the Random Forest model

2.1 Neural Network Model

In this part, we are going to implement the Neural Network model to see how efficient it is with our dataset and our purpose. As in the course we are going to use the MLPRegressor from the sckit learn library.

[77]:
```
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_absolute_error

#Model Creation
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0) #␣
 ↪only partial of the data are used for training.


mlpreg = MLPRegressor(hidden_layer_sizes = [100,200,200,100], #implements the␣
 ↪neural network with it's different parameters
                      activation = 'relu',
                      alpha = 0.01,
                      solver = 'lbfgs').fit(X_train, y_train)

y_pred = mlpreg.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)

#Plots the graph comparing the true and the predicted values.
y_predict_output = mlpreg.predict(X)
```

```
plt.figure()
plt.plot(y[0:230], 'o', label='True values (y_train)',␣
 ↪color='blue',linestyle='dashed',
     linewidth=2)
plt.plot(y_predict_output[0:230], 'x', label='Predictions (y_pred)',␣
 ↪color='red',linestyle='dashed',
     linewidth=2)
plt.title('Comparison of true values and 24-hour predictions')
plt.xlabel('Hours')
plt.ylabel('Rented bike')
plt.legend()
plt.grid()
plt.show()
```



[78]:
```
r2_score(y_test, y_pred) #get the r2 score
```

[78]:
```
0.44423462859972396
```

We can observe that the Neural Network we tested is not very efficient, achieving a score of 0.44. Additionally, it produces some negative values, which are not realistic in real life. One reason for

this result might be the small dataset we used, as Neural Networks typically perform better with larger datasets.

Despite the low score, it was interesting to see how this model behaves with our dataset and to explore its capabilities.

3 Conclusion

In this project, we explored the fundamentals of data science and machine learning using Python, applying the methods covered this semester to the Seoul Bike Sharing Dataset. We eveluated which factors could potentially influence bike demands in Seoul. Throughout our work we investigated correlations between weather conditions and bike rental patterns and predicted values basing on historic data. We analyzed the dataset comprehensively and implemented several predictive models, including linear regression, decision trees, random forests, and boosting algorithms, to predict the bike rental counts. The input features included weather conditions (temperature, humidity, precipitation), temporal variables (seasons, holidays), and other contextual factors.

Key achievements of this work include:

1. Understanding and applying all the methods learned during the year to solve a real-world problem.
2. Comparing the performance of different models in terms of metrics like R2 score, bias, and variance.
3. Highlighting the importance of key input features, such as **temperature** and **hours**, in predicting outcomes.
4. Demonstrating how hyperparameter tuning can significantly influence model performance.

Throughout the whole project we developed our skills majoryly in feature engineering and model evaluation. Predicting rented bike count not only helped us stabilize our theoretical knowledge but also showed challenges that awaits for machine learning researchers.

Among the models tested, the Random Forest algorithm stood out as the best-performing model, offering the highest predictive accuracy and robustness. This suggests that Random Forest could be highly beneficial for optimizing bike stock management or rental availability in Seoul.

While this project remains within an educational context, the insights and methods presented here could be adapted to address real-world challenges, paving the way for data-driven decision-making in urban bike-sharing systems.