Table of Contents

Part 1.1 (Database Management) **3**

Task 1 ..... **3**

Task 2 ..... **3**

Task 3 ..... **3**

Task 4 ..... **3**

Task 5 ..... **4**

Task 6 ..... **4**

Task 7 ..... **4**

Part 1.2 (Data Analytics) **8**

Task 1 ..... **8**

Task 2 ..... **9**

Task 3 ..... **11**

Task 4 ..... **13**

Task 5 ..... **15**

Task 6 ..... **16**

Task 7 ..... **17**

Appendices **19**

Appendix A **19**

Appendix B **21**
Appendix C **22**
Appendix D **23**
Appendix E **24**
Appendix F **25**
Appendix G **26**
Appendix H. Handling null values **36**
Appendix I. Hypothesis Testing **37**
Appendix J. Cyclical/Seasonality **43**
Appendix K. ARIMA model **44**
Appendix L. Random Forest Regressor and Neural Network **46**
Appendix M. Classifiers **48**
Appendix N. Clustering **50**

Part 1.1

Task 1

To create a database using python, first import the sqlite3 library which will enable us to perform database operations, Create, Read, Update and Delete (CRUD). Csv and OS libraries were also imported and will be used in the code.

First, check if a database with a similar name exists. If it those, it gets deleted and the new database gets created. A cursor objected is created and that'll be used to execute database commands.

To create a table using SQLite, use the command "CREATE TABLE *name*…" as shown in Appendix A. The cursor is then used to execute the command. Data contained in the CSV file is then read into the database using the INSERT command. A backup table named "CarSharingBackup" is then created from the original database.

Task 2

To add a column to an already existing database, the "ALTER TABLE … ADD COLUMN…" command is used and executed using the cursor created earlier.

Assigning categorical data to the new column is done by using the "UPDATE" command.

Task 3

A new table "Temperature" is created from the temperature related columns in the original database.
Specific columns are dropped from the CarSharing database using "ALTER TABLE … DROP COLUMN …".

Task 4

The "DISTINCT" keyboard is used to select distinct values from a column. It is used here to get the different weather types in the weather column.
A new column that codes the different weather types is added to the database and then the values in that column are assigned just like in task 2.

Task 5

A Weather table is created from 2 specific columns in the CarSharing database using appropriate commands and executed using the cursor. One of the aforementioned columns is then dropped from the CarSharing database.

Task 6

A table named “Time” is created with four columns containing each row’s timestamp, hour, weekday name, and month name.
strftime() function returns the date formatted according to the format string specified in argument first.
“%H” reads the current hour in the timestamp passed.
“%w” reads the current day of the week in the timestamp passed.
“%m” reads the current month in the timestamp passed as an argument.

Task 7

a. Fetching the date and time with the highest demand rate in 2017 is done by selecting the timestamp and demand column from the CarSharing table where the timestamp contains “2017”, ordering the demand column in descending order and finally, picking the first result by limiting the output to 1 entry.

b. Select the different weekday names and find the average demand for each distinct weekday, all done in the CarSharing table. The CarSharing table is then joined with the Time table to access the year. The data is then grouped and ordered in descending order.
   The first and last rows are the highest and lowest average demand rates.

   A similar pattern is used to obtain the month and the season with the highest and lowest demand rate

c. Select the distinct hours in the CarSharing table and then finding the mean demand for each distinct hour. The CarSharing table is joined with the Time table on the timestamp column using “INNER JOIN”. The weekday with the highest demand (gotten from a previous task) is picked, grouped by hour

and ordered in descending order. A for loop is used to print out the hours with their average demand rate for that weekday.

d. Select the distinct categories from the CarSharing table where the time stamp contains “2017”. Group by the categories and order the number of each group in descending order.

   Select the distinct weathers in the database, joining the time and CarSharing table, selecting the rows that whose data are from 2017, grouping the data by weather code, ordering the grouped data by the number of data points per group in descending and selecting the first data point.

   Select each month and find the mean windspeed for each month. Inner Join with the Time table to obtain those months in the year 2017. Group by month and order by timestamp. A for loop is used to print out the results

   Select months and windspeed in the CarSharing table. The CarSharing table is then joined with the Time table on the timestamp column using “INNER JOIN” to access the year, 2017. The data is then grouped by month and ordered by windspeed in descending order.
   The first and last rows are the months with the highest and lowest windspeed, respectively.

   Select each month and find the mean humidity for each month. Inner Join with the Time table to obtain those months in the year 2017. Group by month and order by timestamp. A for loop is used to print out the results.

   Select months and humidity in the CarSharing table. The CarSharing table is then joined with the Time table on the timestamp column using “INNER JOIN” to access the year, 2017. The data is then grouped by month and ordered by humidity in descending order.
   The first and last rows are the months with the highest and lowest humidity, respectively.

   Select the temperature category and average demand in the year 2017, grouping by temperature category and then ordering the mean demand in

descending order. A for loop is used to print out the result gotten from the database.

e. Since, the month with the highest demand has been obtained in a previous task, select the different temperature categories from CarSharing table and then join the Time and CarSharing table on the timestamp column to get data from 2017 and where the month is the month with the highest demand rate. Data is then grouped by temperature and number of data points in each group is counted and the data is ordered in descending order

   Select the different weathers in CarSharing table then, joining the Time and Weather table with CarSharing using INNERJOIN. Rows that contain 2017 data are selected, grouped by weather code, counted and ordered in descending order. The first row gives the most frequent weather in the month with the highest demand.

   Select the month and average windspeed from CarSharing table, joining the Time and CarSharing table on the timestamp column using INNERJOIN. Selecting those rows that contain 2017 data and where the month equals the month with the highest demand rate.

   The highest and lowest windspeeds in the months with the highest demand rate in 2017 are gotten by querying the databases as follows: select windspeed from the CarSharing table. The CarSharing table is then joined with the Time table on the timestamp column using “INNER JOIN” to access the year 2017, the month with the highest demand and the where the windspeed is not 0. The data is then ordered by windspeed in descending order.
   The first and last rows are the highest and lowest windspeed, respectively.

   Select month and average humidity from CarSharing data, joining the CarSharing and Time table on their time stamp column, selecting the year 2017 and the month with the highest demand.

   The highest and lowest humidity in the months with the highest demand rate in 2017 are gotten by querying the databases as follows: select humidity from the CarSharing table. The CarSharing table is then joined

with the Time table on the timestamp column using “INNER JOIN” to access the year 2017, the month with the highest demand and the where the humidity is not 0. The data is then ordered by humidity in descending order.
The first and last rows are the highest and lowest windspeed, respectively.

Average demand rate by temperature in the month with the highest demand rate in 2017 is gotten by selecting the temperature category, calculating the average demand rate from the CarSharing table, the CarSharing and Time table are joined on the timestamp column using INNERJOIN, data for the year 2017 and the month with the highest demand rate are selected and grouped by the temperature category and ordered by the average demand rate in descending order

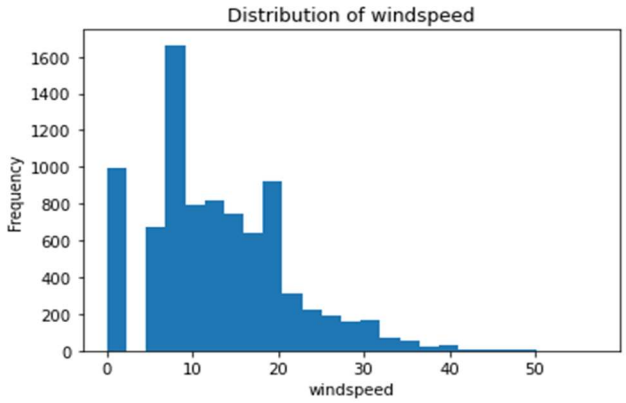
Part 1.2

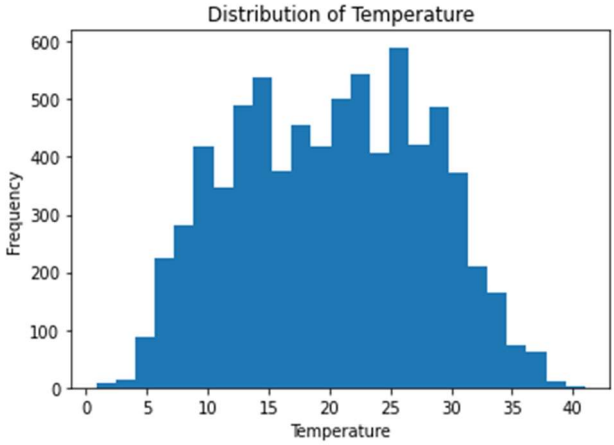
Task 1

N/B: This task uses the codes in appendix H

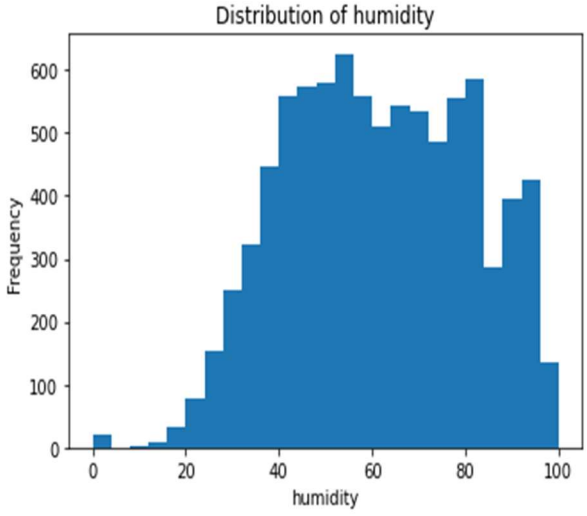
Line 5, checks for duplicate rows and the code returns “0”. This shows that there are no duplicate rows.

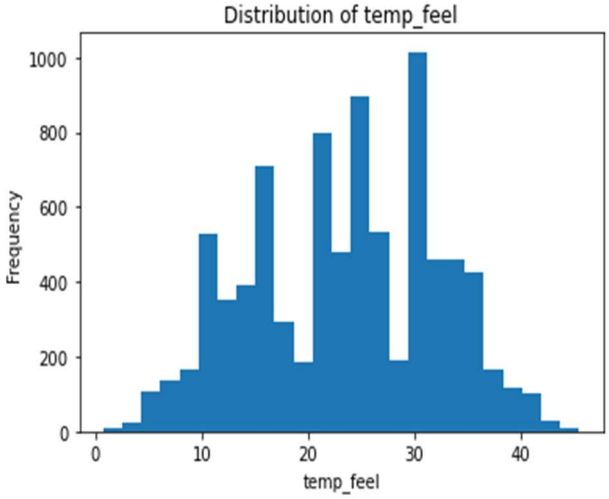
Line 6 checks for the number of null values per column. Results show that the null values per column are less that 15% of the length of the column, therefore, null values can be filled using measures of central tendency





The function in line 7 computes the mean, median and mode of each column as well as plots the distribution of the column





From the diagrams above, the distributions are slightly skewed or tend towards a normal distribution. From the results in line 7, the mode and median are almost equal for all the columns. Therefore, the median value is used to fill the missing values in the dataset as shown in line 8.

The cleaned data is stored as a csv file as shown in line 9.

Task 2

N/B: This task uses the codes in appendix I

Comparing each numerical column against the other: First to know if to use a parametric or a non-parametric method, a normality test function and homogeneity test function are created as shown in line 3 and 4. Outputs of line 5 and 6 show that the columns are neither normally distributed nor have equal variance, therefore, proceed to a non-parametric test.

Mann-Whitney U test is suitable for this analysis.

The Mann-Whitney U test carried out in line 8 shows that,

i. There is a ***statistically significant*** difference between ***temp and temp_feel***
ii. There is a ***statistically significant*** difference between ***temp and humidity***
iii. There is a ***statistically significant*** difference between ***temp and windspeed***
iv. There is a ***statistically significant*** difference between ***temp_feel and humidity***
v. There is a ***statistically significant*** difference between **temp_feel and windspeed**
vi. There is a ***statistically significant*** difference between ***humidity and windspeed***

Comparing each categorical column against the other: This was carried out using chi-square test of independence. The function was created as shown in line 8.

The results are as follows:

i. There's ***a relationship*** between ***season and holiday***
ii. There's ***no relationship*** between ***season and working day***
iii. There's ***a relationship*** between ***season and weather***
iv. There's ***a relationship*** between ***holiday and working day***
v. There's ***a relationship*** between ***holiday and weather***
vi. There's ***a relationship*** between ***working day and weather***

Comparing numerical and categorical columns:

1) *ANOVA*

ANOVA is used to check if there is a significant difference between season and other columns and weather and other columns.

The codes in lines 10 and 11 gave the following result:

i. The relationship between ***season and temp is significant***
ii. The relationship between ***season and temp_feel is significant***
iii. The relationship between ***season and humidity is significant***
iv. The relationship between ***season and windspeed is significant***
v. The relationship between ***weather and temp is significant***
vi. The relationship between ***weather and temp_feel is significant***
vii. The relationship between ***weather and humidity is significant***
viii. The relationship between ***weather and windspeed is significant***

2) T-test

T-test is used to check if there is a significant difference between holiday and other columns and working day and other columns.
Before running a t-test, we have to check for equality of variances.
Line 12 and 14 group the data and show if the 2 columns (Yes/No) have equal variances under temp, temp_feel, humidity and windspeed. The results show that holiday has unequal variances while working day has equal variances.
We already know that each column is independent of the other.
The significance results are as follows:

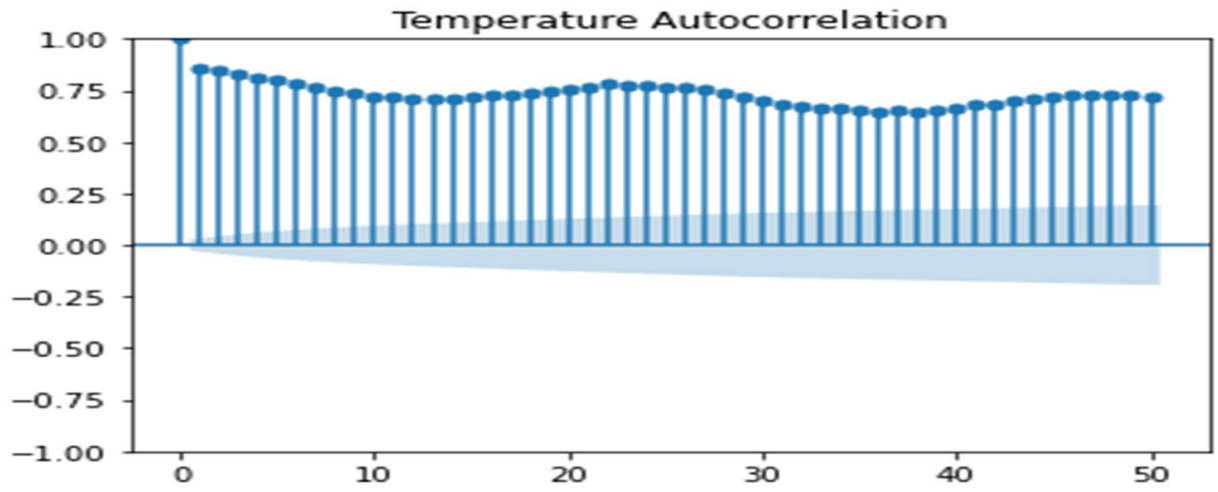
i. The relationship between ***holiday and temp is not significant***
ii. The relationship between ***holiday and temp_feel is not significant***
iii. The relationship between ***holiday and humidity is significant***
iv. The relationship between ***holiday and windspeed is not significant***
v. The relationship between ***working day and temp is significant***
vi. The relationship between ***working day and temp_feel is significant***
vii. The relationship between ***working day and humidity is not significant***
viii. The relationship between ***working day and windspeed is not significant***

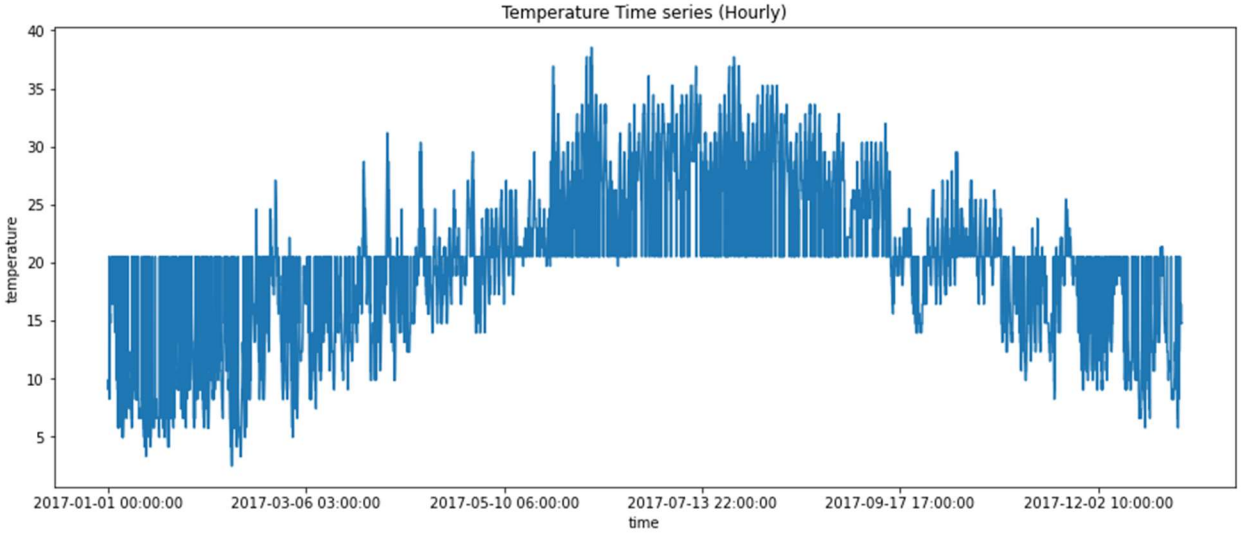
Task 3

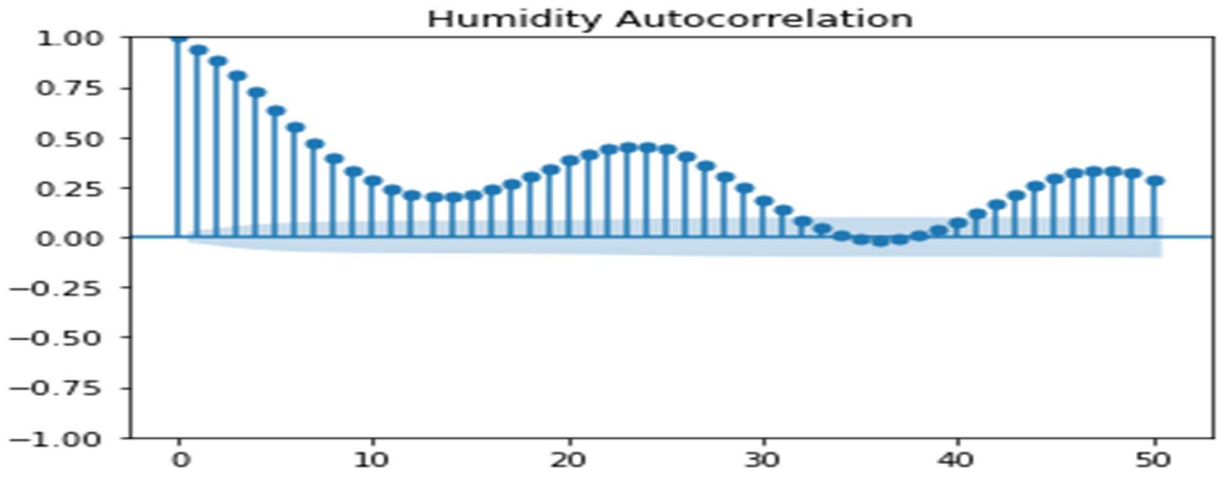
***N/B*: This task uses the codes in appendix J**

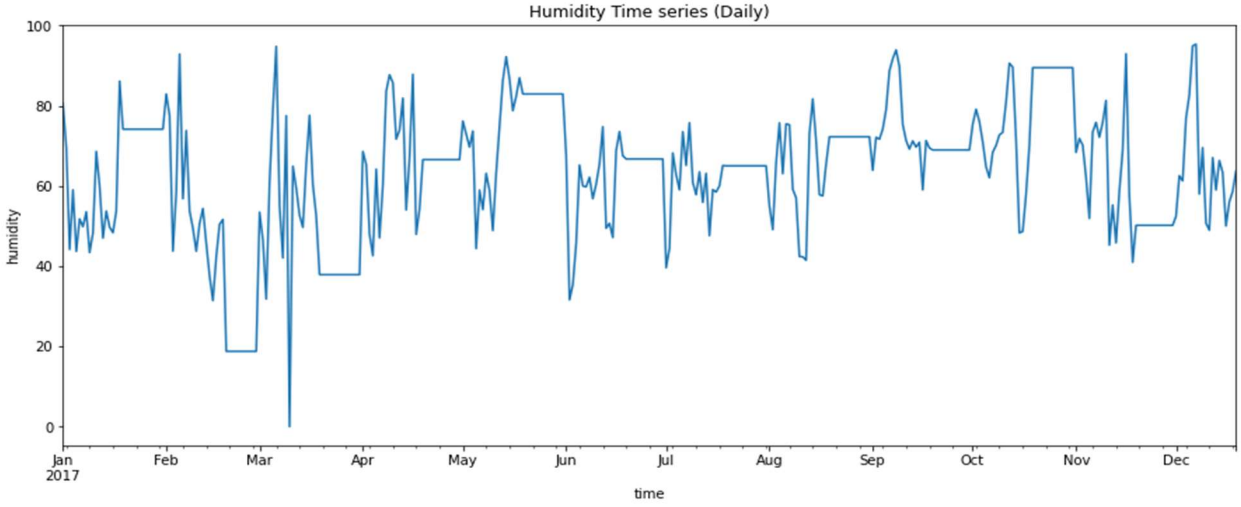
To determine the kind of trend/pattern contained in each column, the distribution of the specified columns are plotted out. Some of the columns were converted to daily data to get a clearer picture of what the data has to say

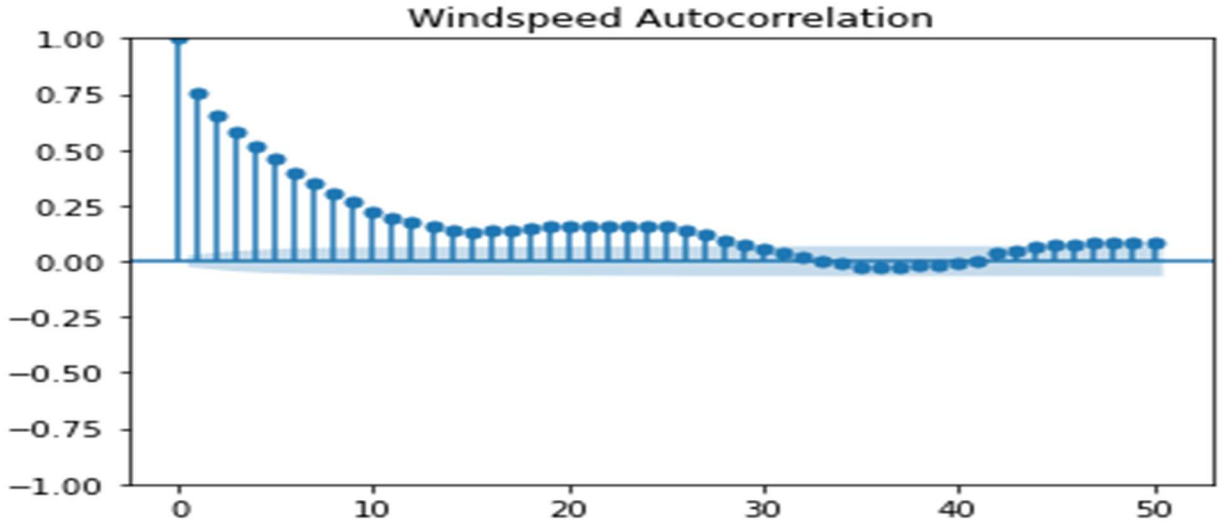
The codes in lines 4,5,6 and 7 gave the following plots:

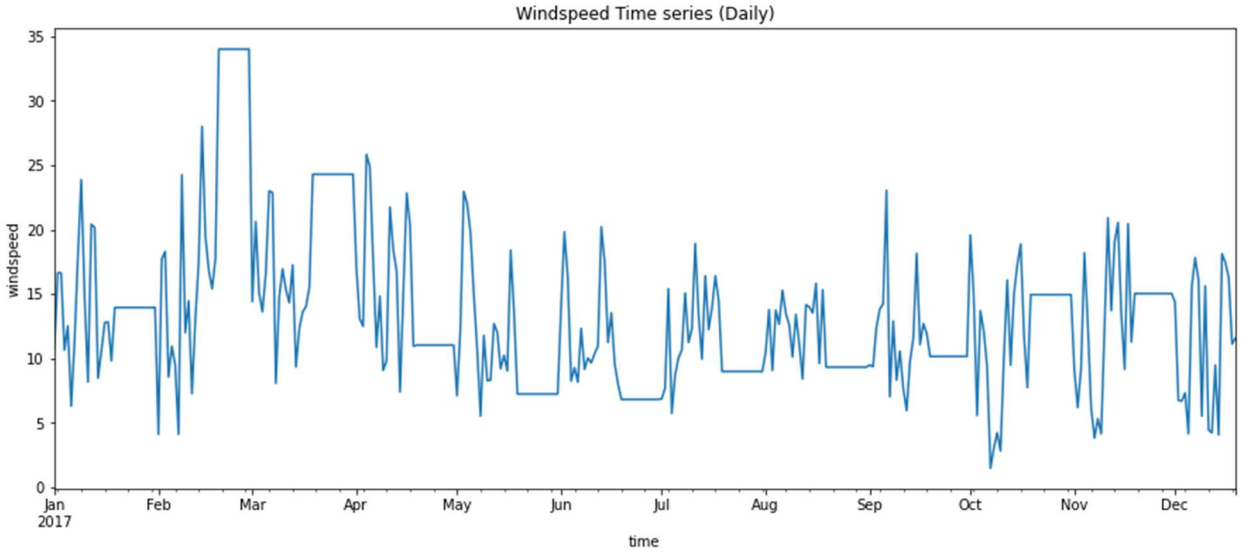


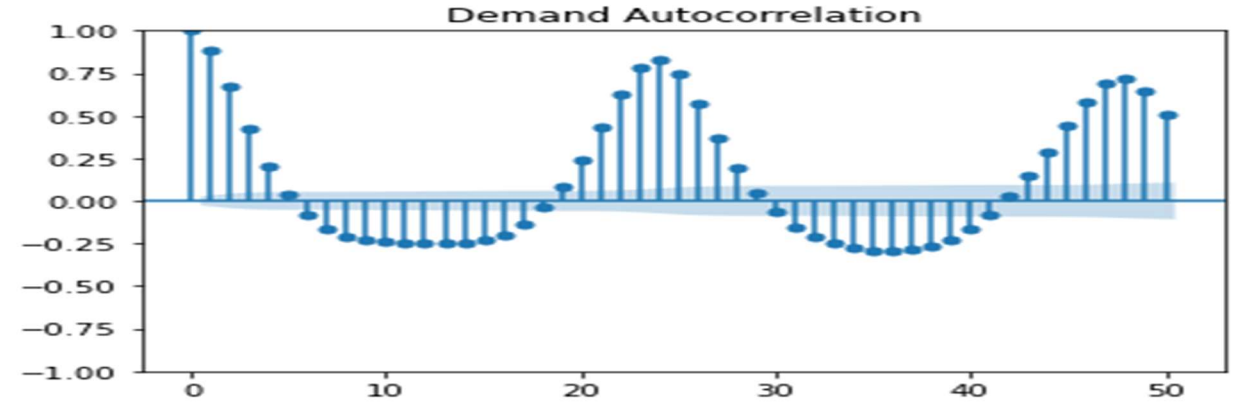


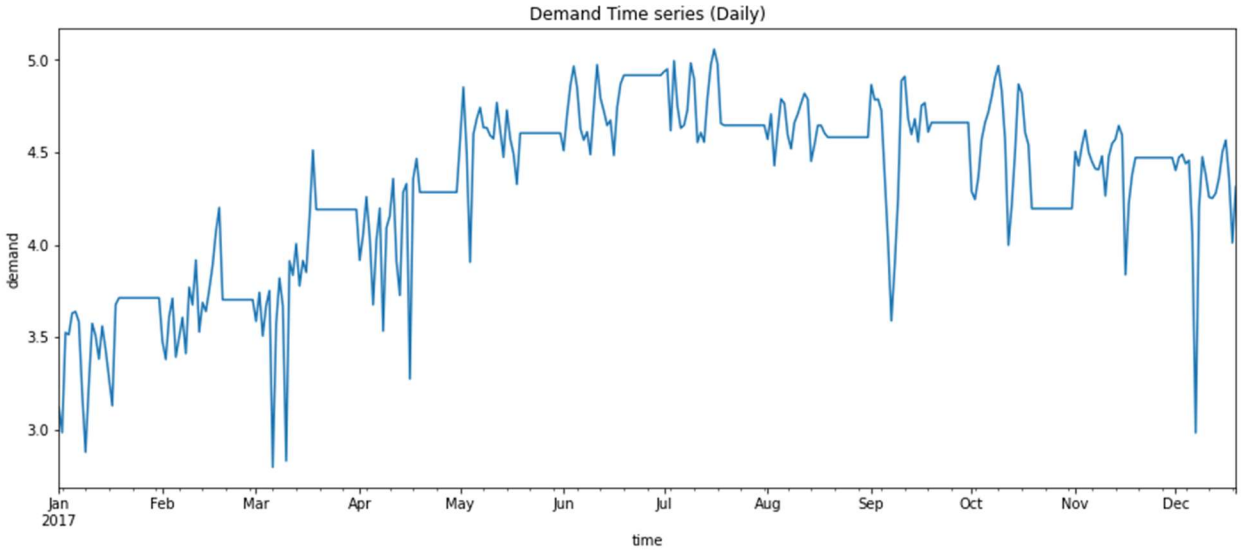

Humidity Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50


Humidity Time series (Daily)
humidity
100
80
60
40
20
0
Jan
2017
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
time


Windspeed Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50


Windspeed Time series (Daily)
windspeed
time
Jan
2017
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec


Demand Autocorrelation


Demand Time series (Daily)
demand
time
Jan
2017
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec

Temperature: There's a clear cyclical pattern in the plot of the time series of temperature and little seasonality is depicted from the ACF plot

Humidity: There's monthly variations in the humidity data. From the plot, it can be seen that clusters are formed in each month of the year. The ACF plot shows oscillations, therefore, there is seasonlity.

Wind speed: There's monthly seasonality in the windspeed data. From the time series plot, it can be seen that clusters are formed in each month of the year. Also, the ACF plot shows some level of oscillation. However, there is no cyclical variation.

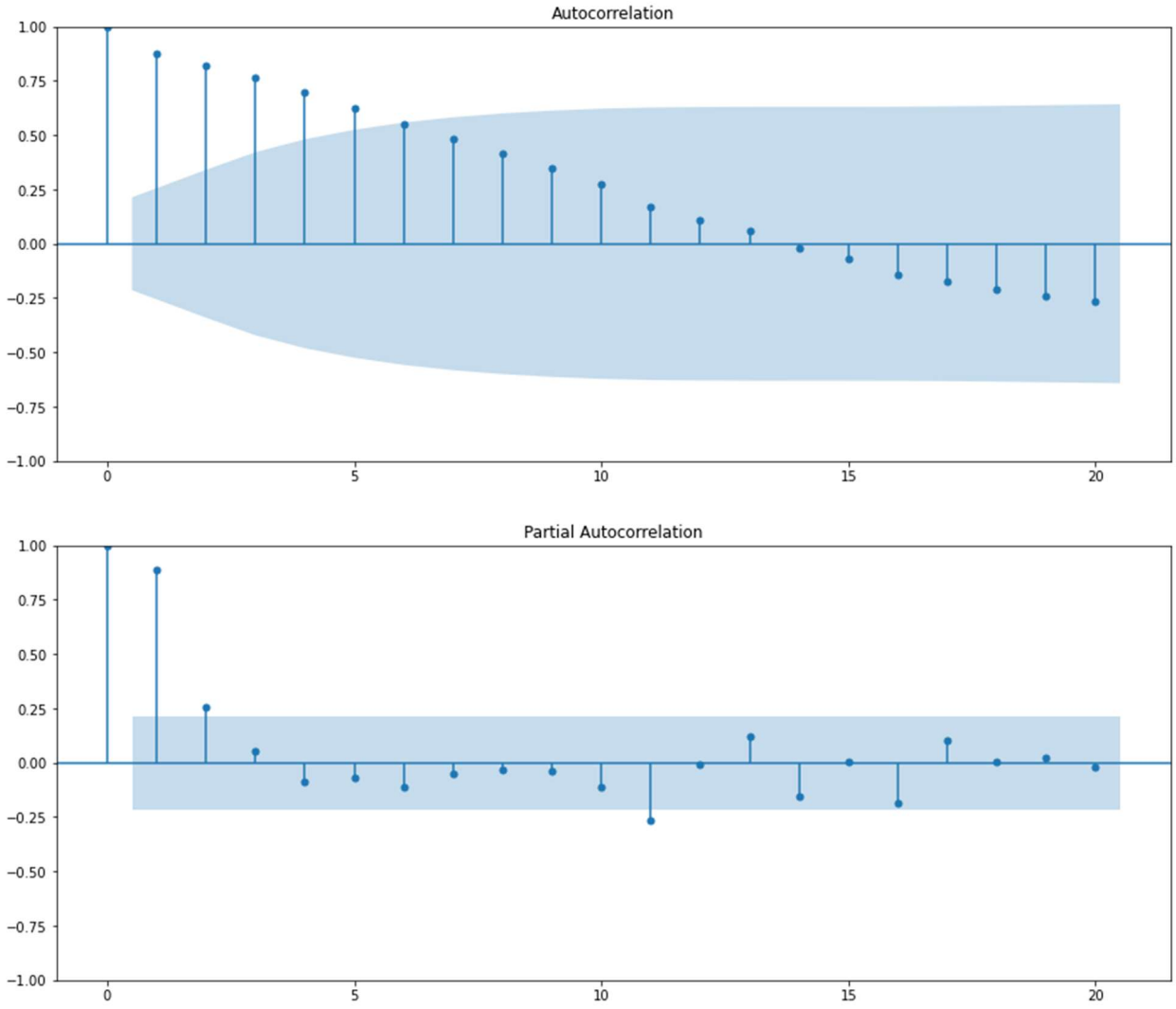
Demand: From the time series and ACF plot above, it is observed that there's monthly seasonality in the data as clusters are being formed in each month of the year.

Task 4

***N/B*: This task uses the codes in Appendix K**

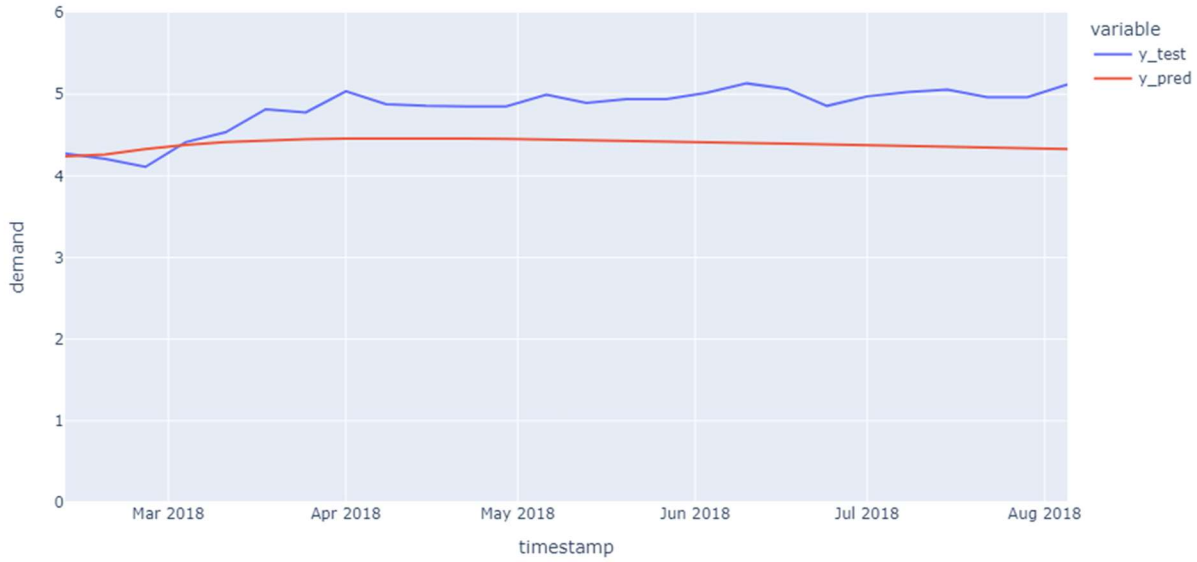
After importing the necessary packages and reading in the data (setting the timestamp as index), the data was then resampled to weekly data using method of *mean()* as shown in line 3. This was done to predict the *weekly average* demand rate.

To determine the parameters to use, an ACF and a PACF plot were made using the codes at lines 4 and 5 respectively and are show below:



The significant values for the ACF plot are 0,1,2,3,4,5 while the significant values for the PACF plot are 0,1,2,11. These values were then used as hyperparameters to determine the best model as shown in line 12. p = 2 and q = 3 gave the lowest MAE (0.4558). Those values were used to build the ARIMA model and make predictions as shown in line 13 and 14.

Line 15 plots a line chart showing the original values and the predicted values as shown below:



Task 5

N/B: This task uses the codes in Appendix L

After importing the necessary libraries and reading in the data, the timestamp column was dropped because it is not needed for this data analysis. The categorical variables were encoded using get_dummies function in the pandas library (line 3). The newly created columns were renamed and unnecessary columns were dropped (line 4). The data was split vertically, feature columns and target column and then horizontally, to get a training set and a test set (line 5 and 6). 20% of the data was used for testing

1. Random Forest Regressor:
   The Random Forest Regressor model is created and fit on the training data as shown in line 7. The model is used to predict the demand rate (line 8). Line 9, the Mean Squared Error is calculated using the original values from the dataset against the values predicted by the model. Line 10 prints the output of the calculation, which is **1.3989.**

2. Deep Neural Network:
   An MLPRegressor is created (line 11). The model is fitted in the training data and predictions are being made as shown in line 12. The MSE for the model was then calculated using at line 13 and the value gotten was **1.5134.**

   The Mean Squared Error measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value.

   Judging from the MSE of both models**, the Random Forest Regressor gives a lower MSE**, therefore, the **Random Forest Regressor performed better**

Task 6

N/B: This task uses the codes in Appendix M

To categorize the data into two groups, one group containing demand whose values are more that the average demand and the other group, those whose values are less than the average demand.

First, the average demand is obtained at line 3, then a binary column is created at line 4 and 5. The categorical variables were encoded using get_dummies function in the pandas library (line 6). The newly created columns were renamed and unnecessary columns were dropped (line 7). The data was split vertically, feature columns and target column and then horizontally, to get a training set and a test set (line 8 and 9). 30% of the data was used for testing as per the instructions.

The classifiers used for this task are:

i. Logistic Regression Classifier
ii. Decision Tree Classifier
iii. Gradient Boosting Classifier

The accuracy gotten from the code at line 10, 11 and 12 are 1.0, 1.0 and 1.0 respectively.

Task 7

N/B: This task uses the codes in Appendix N

The necessary columns needed for the analysis were read into a dataframe. The dataframe was split such that it only contains temperature data in 2017 (line 3). A scalar is used to scale the variables so that they can be easily compared and plotted (line 4 and 5). The scaled data is then used for the models.

KMeans: Line 7 is a function that creates a kmeans model based on the number of clusters passed into the function, fits the model on the data and then makes predictions.

Gaussian Mixture Distribution: Line 9 is a function that creates a Gaussian model based on the number of clusters passed into the function, fits the model on the data and then makes predictions.

2 clusters KMeans gives the best Uniform cluster but 2 cluster Gaussian does not give the best Uniform clusters. On the other hand, 4 cluster Gaussian and KMeans give very good uniform clusters. **Therefore, the k with the most uniform cluster is k=4.**

Appendix A

```
import sqlite3
import csv
import os

# delete database file if it already exists
if os.path.exists("car_sharing_database"):
    os.remove("car_sharing_database")

#  task 1 create database
connection = sqlite3.connect("car_sharing_database")

#  task 1: An object that'll allow us to query the database
cursor = connection.cursor()

try:
    #  task 1 create CarSharing table
    car_sharing_table_creation = """CREATE TABLE CarSharing (
    id INT,
    timestamp TEXT,
    season TEXT,
    holiday TEXT,
    workingday TEXT,
    weather TEXT,
    temp REAL,
    temp_feel REAL,
    humidity REAL,
    windspeed REAL,
    demand REAL
    )"""

      # Executing the command above
    connection.execute(car_sharing_table_creation)
    print("CarSharing table created successfully")
except:
    print("CarSharing table already exists")

print()
```

```
try:
    #   task 1 import csv file and read into database
    clear_car_sharing_table = "DELETE FROM CarSharing"
    cursor.execute(clear_car_sharing_table)
    csv_file = open("CarSharing.csv") # Opens csv file
    csv_file_rows = csv.reader(csv_file) # Reads each row of the csv file
    car_sharing_insert_query = """INSERT INTO CarSharing (id, timestamp, season,
holiday, workingday, weather, temp, temp_feel,
        humidity, windspeed, demand) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"""
      # Inserts data from the csv file into the database
    cursor.executemany(car_sharing_insert_query, csv_file_rows)

    print("CSV file imported to CarSharing table")
except:
    print("CarSharing table already populated")

print()

try:
    #   task 1 create backup table
    car_sharing_backup = "CREATE TABLE CarSharingBackup SELECT * FROM CarSharing"
    cursor.execute(car_sharing_backup)
    print("Backup table created successfully")
except:
    print("Backup table already exists")
```

Appendix B

```
try:
    #   task 2 add temperature category column
    temperature_category_alter = "ALTER TABLE CarSharing ADD COLUMN temp_category
TEXT"
    cursor.execute(temperature_category_alter)
    print("Temperature category column added successfully")
except:
    print("Temperature category column already exists")

print()

try:
    #   task 2 update temperature category based on temperature feel like value
    temperature_category_update = """
    UPDATE CarSharing SET temp_category = "Cold" WHERE temp_feel < 10;
    UPDATE CarSharing SET temp_category = "Mild" WHERE temp_feel >= 10 AND
temp_feel <= 25;
    UPDATE CarSharing SET temp_category = "Hot" WHERE temp_feel > 25;
    """
    cursor.executescript(temperature_category_update)
    print("Temperature category updated successfully")
except:
    print("Temperature category already updated")

print()
```

Appendix C

```
try:
    #   task 3 create temperature table
    temperature_creation = "CREATE TABLE Temperature SELECT temp, temp_feel,
temp_category FROM CarSharing"
    cursor.execute(temperature_creation)
    print("Temperature table created successfully")
except:
    print("Temperature table already exists")

print()

try:
    #   task 3 drop temperature and temperature feel like columns
    temperature_drop = """
    ALTER TABLE CarSharing DROP COLUMN temp;
    ALTER TABLE CarSharing DROP COLUMN temp_feel;
    """
    cursor.executescript(temperature_drop)
    print("Temperature and temperature feel like columns dropped successfully")
except:
    print("Temperature and temperature feel like columns already dropped")

print()
```

Appendix D

```
try:
    #   task 4 find distinct values of weather column
    distinct_weather_query = "SELECT DISTINCT weather FROM CarSharing"
    weathers = cursor.execute(distinct_weather_query).fetchall()
    print("Distinct values of weather column are:")
    for distinct_weather in weathers:
        weather = distinct_weather[0]
        print(weather)
except:
    print("An error occurred while finding distinct values of weather column")

print()

try:
    #   task 4 add weather code column
    weather_code_alter = "ALTER TABLE CarSharing ADD COLUMN weather_code INT"
    cursor.execute(weather_code_alter)
    print("Weather code column added successfully")
except:
    print("Weather code column already exists")

print()

try:
    #   task 4 update values of weather code column
    weather_code_update = """
    UPDATE CarSharing SET weather_code = 1 WHERE weather LIKE "%cloudy%";
    UPDATE CarSharing SET weather_code = 2 WHERE weather LIKE "%mist%";
    UPDATE CarSharing SET weather_code = 3 WHERE weather LIKE "%light%";
    UPDATE CarSharing SET weather_code = 4 WHERE weather LIKE "%heavy%";
    """
    cursor.executescript(weather_code_update)
    print("Weather code column updated successfully")
except:
    print("Weather code column already updated")

print()
```

Appendix E

```
try:
    #   task 5 create weather table
    weather_creation = "CREATE TABLE Weather AS SELECT weather, weather_code FROM
CarSharing"
    cursor.execute(weather_creation)
    print("Weather table created successfully")
except:
    print("Weather table already exists")

print()

try:
    #   task 5 drop weather column
    weather_drop = "ALTER TABLE CarSharing DROP COLUMN weather"
    cursor.execute(weather_drop)
    print("Weather column dropped successfully")
except:
    print("Weather column already dropped")

print()
```

Appendix F

```
try:
    #   task 6 create time table
    time_creation = """
    CREATE TABLE Time AS
    SELECT timestamp,
           strftime("%H", timestamp) AS hour,
           CASE cast(strftime('%w', timestamp) as INT)
               WHEN 0 THEN 'Sunday'
               WHEN 1 THEN 'Monday'
               WHEN 2 THEN 'Tuesday'
               WHEN 3 THEN 'Wednesday'
               WHEN 4 THEN 'Thursday'
               WHEN 5 THEN 'Friday'
               WHEN 6 THEN 'Saturday'
           END AS weekday,
           CASE cast(strftime('%m', timestamp) as INT)
               WHEN 1 THEN 'January'
               WHEN 2 THEN 'February'
               WHEN 3 THEN 'March'
               WHEN 4 THEN 'April'
               WHEN 5 THEN 'May'
               WHEN 6 THEN 'June'
               WHEN 7 THEN 'July'
               WHEN 8 THEN 'August'
               WHEN 9 THEN 'September'
               WHEN 10 THEN 'October'
               WHEN 11 THEN 'November'
               WHEN 12 THEN 'December'
           END AS month
        FROM CarSharing
    """
    cursor.execute(time_creation)
    print("Time table created successfully")
except:
    print("Time table already exists")

print()
```

Appendix G

```
try:
    #   task 7a select date and time with highest demand rate in 2017
    highest_demand_rate_query = """
    SELECT timestamp, demand FROM CarSharing
    WHERE timestamp LIKE "2017%"
    ORDER BY demand DESC
    LIMIT 1
    """
    result = cursor.execute(highest_demand_rate_query)
    timestamp, demand = result.fetchone()
    print(f"Date and time with highest demand rate in 2017 was {timestamp} with
demand rate of {demand}")
except:
    print("An error occurred while finding date and time with highest demand rate
in 2017")

print()

#   task 7b select weekday with the highest and lowest average demand rates in
2017
try:
    demand_rate_weekday_query = """
    SELECT DISTINCT weekday, AVG(demand) AS average_demand_rate FROM CarSharing c
    INNER JOIN Time t ON c.timestamp = t.timestamp
    WHERE t.timestamp LIKE "2017%"
    GROUP BY weekday
    ORDER BY average_demand_rate DESC
    """
    result = cursor.execute(demand_rate_weekday_query)
    rows = result.fetchall()
    highest_weekday, highest_average_demand = rows[0]
    lowest_weekday, lowest_average_demand = rows[-1]
    print(f"Weekday with highest average demand rate in 2017 was
{highest_weekday} with average demand rate of {highest_average_demand}")
    print(f"Weekday with lowest average demand rate in 2017 was {lowest_weekday}
with average demand rate of {lowest_average_demand}")
except Exception as e:
    print(e)
    print("An error occurred while finding weekday with the highest and lowest
average demand rates in 2017")

print()
```

```
#   task 7b select month with the highest and lowest average demand rates in 2017
try:
    demand_rate_month_query = """
    SELECT DISTINCT month, AVG(demand) AS average_demand_rate FROM CarSharing c
    INNER JOIN Time t ON c.timestamp = t.timestamp
    WHERE t.timestamp LIKE "2017%"
    GROUP BY month
    ORDER BY average_demand_rate DESC
    """
    result = cursor.execute(demand_rate_month_query)
    rows = result.fetchall()
    highest_month, highest_average_demand = rows[0]
    lowest_month, lowest_average_demand = rows[-1]
    print(f"Month with highest average demand rate in 2017 was {highest_month}
with average demand rate of {highest_average_demand}")
    print(f"Month with lowest average demand rate in 2017 was {lowest_month} with
average demand rate of {lowest_average_demand}")
except Exception as e:
    print(e)
    print("An error occurred while finding month with the highest and lowest
average demand rates in 2017")

print()

#   task 7b select season with the highest and lowest average demand rates in
2017
try:
    demand_rate_season_query = """
   SELECT DISTINCT season, AVG(demand) AS average_demand_rate FROM CarSharing
    WHERE timestamp LIKE "2017%"
    GROUP BY season
    ORDER BY average_demand_rate DESC
    """
    result = cursor.execute(demand_rate_season_query)
    rows = result.fetchall()
    highest_season, highest_average_demand = rows[0]
    lowest_season, lowest_average_demand = rows[-1]
    print(f"Season with highest average demand rate in 2017 was {highest_season}
with average demand rate of {highest_average_demand}")
    print(f"Season with lowest average demand rate in 2017 was {lowest_season}
with average demand rate of {lowest_average_demand}")
except Exception as e:
    print(e)
    print("An error occurred while finding season with the highest and lowest
average demand rates in 2017")
```

```
#   task 7c select hours of weekday with highest average demand rate in 2017
try:
    demand_rate_hours_query = f"""
    SELECT DISTINCT hour, AVG(demand) AS average_demand_rate FROM CarSharing c
    INNER JOIN Time t ON c.timestamp = t.timestamp
    WHERE weekday = "{highest_weekday}" AND t.timestamp LIKE "2017%"
    GROUP BY hour
    ORDER BY average_demand_rate DESC    """
    result = cursor.execute(demand_rate_hours_query)
    rows = result.fetchall()

    print(f"Hours of {highest_weekday} with highest average demand rate in 2017
are:")
    for current_row in rows:
        hour, average_demand = current_row
        print(f"Hour: {hour} | Average demand rate: {average_demand}")

except Exception as e:
    print(e)
    print("An error occurred while finding hours of weekday with highest average
demand rate in 2017")

print()

#   task 7d select most frequent temperature in 2017
try:
    most_frequent_temperature_query = """
    SELECT DISTINCT temp_category most_frequent_temperature FROM CarSharing
    WHERE timestamp LIKE "2017%"
    GROUP BY temp_category
    ORDER BY count(*) DESC
    LIMIT 1
    """
    result = cursor.execute(most_frequent_temperature_query)
    most_frequent_temperature = result.fetchone()[0]
    print(f"Most frequent temperature in 2017 was {most_frequent_temperature}")
except:
    print("An error occurred while finding most frequent temperature in 2017")

print()
```

```
#   task 7d select most frequent weather in 2017
try:
    most_frequent_weather_query = """
    SELECT DISTINCT w.weather most_frequent_weather FROM CarSharing c
    INNER JOIN Weather w on c.weather_code = w.weather_code
    WHERE timestamp LIKE "2017%"
    GROUP BY c.weather_code
    ORDER BY count(c.weather_code) DESC
    LIMIT 1
    """
    result = cursor.execute(most_frequent_weather_query)
    most_frequent_weather = result.fetchone()[0]
    print(f"Most frequent weather in 2017 was {most_frequent_weather}")
except Exception as e:
    print(e)
    print("An error occurred while finding most frequent weather in 2017")

print()

#   task 7d select average windspeed by months in 2017
try:
    average_windspeed_query = """
    SELECT month, AVG(windspeed) average_windspeed FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%"
    GROUP BY month
    ORDER BY c.timestamp
    """
    result = cursor.execute(average_windspeed_query)
    rows = result.fetchall()

    print("Average windspeed by months in 2017:")

    for current_row in rows:
        month, average_windspeed = current_row
        print(f"Month: {month} | Average windspeed: {average_windspeed}")
except Exception as e:
    print(e)
    print("An error occurred while finding average windspeed by months in 2017")

print()
```

```
#   task 7d select highest and lowest windspeeds in 2017
try:
    windspeed_query = """
   SELECT month, windspeed FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%"
    GROUP BY month
    ORDER BY windspeed DESC
    """
    result = cursor.execute(windspeed_query)
    rows = result.fetchall()
    highest_windspeed_month, highest_windspeed = rows[0]
    lowest_windspeed_month, lowest_windspeed = rows[-1]
    print(f"Month with highest windspeed in 2017 was {highest_windspeed_month}
with windspeed of {highest_windspeed}")
    print(f"Month with lowest windspeed in 2017 was {lowest_windspeed_month} with
windspeed of {lowest_windspeed}")
except Exception as e:
    print(e)
    print("An error occurred while finding highest and lowest windspeeds in
2017")

print()

#   task 7d select average humidity by months in 2017
try:
    average_humidity_query = """
    SELECT month, AVG(humidity) average_humidity FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%"
    GROUP BY month
    ORDER BY c.timestamp
    """
    result = cursor.execute(average_humidity_query)
    rows = result.fetchall()

    print("Average humidity by months in 2017:")

    for current_row in rows:
        month, average_humidity = current_row
        print(f"Month: {month} | Average humidity: {average_humidity}")
except Exception as e:
    print(e)
    print("An error occurred while finding average humidity by months in 2017")
```

```
#   task 7d select highest and lowest humidities in 2017
try:
    humidity_query = """
   SELECT month, windspeed FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%"
    GROUP BY month
    ORDER BY windspeed DESC
    """
    result = cursor.execute(humidity_query)
    rows = result.fetchall()
    highest_humidity_month, highest_humidity = rows[0]
    lowest_humidity_month, lowest_humidity = rows[-1]
    print(f"Month with highest humidity in 2017 was {highest_humidity_month} with
humidity of {highest_humidity}")
    print(f"Month with lowest humidity in 2017 was {lowest_humidity_month} with
humidity of {lowest_humidity}")
except Exception as e:
    print(e)
    print("An error occurred while finding highest and lowest windspeeds in
2017")

print()

#   task 7d select average demand rate by temperature in 2017
try:
    average_demand_rate_by_temperature_query = """
    SELECT temp_category, AVG(demand) average_demand_rate FROM CarSharing c
    WHERE timestamp LIKE "2017%"
    GROUP BY temp_category
    ORDER BY average_demand_rate DESC
    """
    result = cursor.execute(average_demand_rate_by_temperature_query)
    rows = result.fetchall()

    print("Average demand rates by temperature in 2017:")

    for current_row in rows:
        temperature, average_demand = current_row
        print(f"Temperature: {temperature} | Average demand rate:
{average_demand}")
except Exception as e:
    print(e)
    print("An error occurred while finding average demand rates by temperature in
2017")
```

```
#   task 7e select most frequent temperature in month with highest demand rate in
2017
try:
    most_frequent_temperature_query = f"""
    SELECT DISTINCT temp_category most_frequent_temperature FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%" AND month = "{highest_month}"
    GROUP BY temp_category
    ORDER BY count(*) DESC
    LIMIT 1
    """
    result = cursor.execute(most_frequent_temperature_query)
    most_frequent_temperature = result.fetchone()[0]
    print(f"Most frequent temperature in {highest_month}, 2017 was
{most_frequent_temperature}")
except:
    print(f"An error occurred while finding most frequent temperature in
{highest_month}, 2017")

print()

#   task 7e select most frequent weather in month with highest demand rate in
2017
try:
    most_frequent_weather_query = f"""
    SELECT DISTINCT w.weather most_frequent_weather FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    INNER JOIN Weather w on c.weather_code = w.weather_code
    WHERE c.timestamp LIKE "2017%" AND month = "{highest_month}"
    GROUP BY c.weather_code
    ORDER BY count(c.weather_code) DESC
    LIMIT 1
    """
    result = cursor.execute(most_frequent_weather_query)
    most_frequent_weather = result.fetchone()[0]
    print(f"Most frequent weather in {highest_month}, 2017 was
{most_frequent_weather}")
except Exception as e:
    print(e)
    print(f"An error occurred while finding most frequent weather in
{highest_month}, 2017")
```

```
#   task 7e select average windspeed in month with highest demand rate in 2017
try:
    average_windspeed_query = f"""
    SELECT month, AVG(windspeed) average_windspeed FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%" AND month = "{highest_month}"
    GROUP BY month    """
    result = cursor.execute(average_windspeed_query)
    rows = result.fetchall()

    print(f"Average windspeed in {highest_month}, 2017:")

    for current_row in rows:
        month, average_windspeed = current_row
        print(f"Month: {month} | Average windspeed: {average_windspeed}")
except Exception as e:
    print(e)
    print(f"An error occurred while finding average windspeed in {highest_month},
2017")

print()

#   task 7e select highest and lowest windspeeds in month with highest demand
rate in 2017
try:
    windspeed_query = f"""
   SELECT windspeed FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%" AND month = "{highest_month}" AND
length(windspeed) > 0
    ORDER BY windspeed DESC
    """
    result = cursor.execute(windspeed_query)
    rows = result.fetchall()
    highest_windspeed = rows[0][0]
    lowest_windspeed = rows[-1][0]
    print(f"Highest windspeed in {highest_month}, 2017 was {highest_windspeed}")
    print(f"Lowest windspeed in {highest_month}, 2017 was {lowest_windspeed}")
except Exception as e:
    print(e)
    print(f"An error occurred while finding highest and lowest windspeeds in
{highest_month}, 2017")
```

```
#   task 7e select average humidity in month with highest demand rate in 2017
try:
    average_humidity_query = f"""
    SELECT month, AVG(humidity) average_humidity FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%" AND month = "{highest_month}"
    GROUP BY month
    """
    result = cursor.execute(average_humidity_query)
    rows = result.fetchall()

    print(f"Average humidity in {highest_month}, 2017:")

    for current_row in rows:
        month, average_humidity = current_row
        print(f"Month: {month} | Average humidity: {average_humidity}")
except Exception as e:
    print(e)
    print(f"An error occurred while finding average humidity in {highest_month},
2017")

print()

#   task 7e select highest and lowest humidities in month with highest demand
rate in 2017
try:
    humidity_query = f"""
    SELECT  humidity FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%" AND month = "{highest_month}" AND
length(humidity) > 0
    ORDER BY humidity DESC
    """
    result = cursor.execute(humidity_query)
    rows = result.fetchall()
    highest_humidity = rows[0][0]
    lowest_humidity = rows[-1][0]
    print(f"Highest humidity in 2017 was {highest_humidity}")
    print(f"Lowest humidity in 2017 was {lowest_humidity}")

except Exception as e:
    print(e)
    print(f"An error occurred while finding highest and lowest windspeeds in
{highest_month}, 2017")
```

```
#   task 7e select average demand rate by temperature in month with highest
demand rate in 2017
try:
    average_demand_rate_by_temperature_query = f"""
    SELECT temp_category, AVG(demand) average_demand_rate FROM CarSharing c
    INNER JOIN Time t on c.timestamp = t.timestamp
    WHERE c.timestamp LIKE "2017%" AND month = "{highest_month}"
    GROUP BY temp_category
    ORDER BY average_demand_rate DESC
    """
    result = cursor.execute(average_demand_rate_by_temperature_query)
    rows = result.fetchall()

    print(f"Average demand rates by temperature in {highest_month}, 2017:")

    for current_row in rows:
        temperature, average_demand = current_row
        print(f"Temperature: {temperature} | Average demand rate:
{average_demand}")
except Exception as e:
    print(e)
    print(f"An error occurred while finding average demand rates by temperature
in {highest_month}, 2017")

print()

connection.commit()
connection.close()

print("Database management tasks have successfully been completed")
```

Appendix H (Handling Null Values)

```
# Import libraries
   1. import pandas as pd
   2. import numpy as np
   3. from matplotlib import pyplot as plt

# Import data and sets the "id" column as the index of the dataset
   4. df = pd.read_csv("car_sharing.csv",index_col = "id")

# duplicated() return True if a row is duplicated, sum() adds up all the "True"
values
   5. df.duplicated().sum()

# Checks for null values per column
   6. df.isnull().sum()

# A function that plots the distribution of a column and computes the mean,
median and mode of a column
   7. def compute_hist_central_tendency(col):
      #plot histogram
       plt.hist(df[col], bins = 25)
      # add labels and title
       plt.ylabel("Frequency")
       plt.xlabel(col)
       plt.title("Distribution of "+ col)

      # print out values
       print(col, "mean: ", df[col].mean())
       print(col, "median: ", df[col].median())
       print(col, "mode: ", df[col].mode())

   8. # filling missing values with medians of the respective columns
      df['temp'].fillna(df['temp'].median(), inplace = True)
      df['humidity'].fillna(df['humidity'].median(), inplace = True)
      df['windspeed'].fillna(df['windspeed'].median(), inplace = True)
      df['temp_feel'].fillna(df['temp_feel'].median(), inplace = True)

# Save cleaned data
   9. df.to_csv("car_sharing_cleaned")
```

Appendix I (Hypothesis Testing)

```
1.  # Import libraries
import pandas as pd
from matplotlib import pyplot as plt
from scipy import stats

from statsmodels.stats.contingency_tables import Table2x2
from statsmodels.stats.power import GofChisquarePower

import statsmodels.api as sm


2.  # Read csv data into a dataframe and inspect data
df = pd.read_csv("car_sharing_cleaned",index_col = "id" )
df.head()
df.info()
```

Numerical vs Numerical significance test

```
    # Checks if data is normally distributed
3.  def check_normality(data):
       test_stat_normality, p_value_normality=stats.shapiro(data)
       print("p value:%.4f" % p_value_normality)
       if p_value_normality <0.05:
           print("Reject null hypothesis >> The data is not normally
       distributed")
       else:
           print("Fail to reject null hypothesis >> The data is normally
       distributed")

    # Checks if two columns have the same variance
4.  def check_variance_homogeneity(group1, group2):
       test_stat_var, p_value_var= stats.levene(group1,group2)
       print("p value:%.4f" % p_value_var)
       if p_value_var <0.05:
           print("Reject null hypothesis >> The variances of the samples are
          different.")
       else:
           print("Fail to reject null hypothesis >> The variances of the
          samples are same.")
```

```
   5. Using line 3 to check for normality in columns
check_normality(df["temp"])
check_normality(df["temp_feel"])
check_normality(df["humidity"])
check_normality(df["windspeed"])

   6. Using line 4 to check for homogeneity of variances
check_variance_homogeneity(df["temp"], df["temp_feel"])
check_variance_homogeneity(df["temp"], df["humidity"])
check_variance_homogeneity(df["temp"], df["windspeed"])
check_variance_homogeneity(df["temp_feel"], df["humidity"])
check_variance_homogeneity(df["temp_feel"], df["windspeed"])
check_variance_homogeneity(df["humidity"], df["windspeed"])

Running mann-whitney U test
   7.  def mannwhitney_test(col1, col2):
         ttest,pvalue = stats.mannwhitneyu(df[col1],df[col2], alternative="two-
            sided")
         print("p-value:%.4f" % pvalue)
         if pvalue <0.05:
             print("Reject null hypothesis >> it can be said that there is a
            statistically significant difference between",
                   col1, "and",col2)
         else:
             print("Fail to reject null hypothesis")
   8. Using function in 7 to run mann-whitney u test
mannwhitney_test("temp", "temp_feel")
mannwhitney_test("temp", "humidity")
mannwhitney_test("temp", "windspeed")
mannwhitney_test("temp_feel", "humidity")
mannwhitney_test("temp_feel", "windspeed")
mannwhitney_test("humidity", "windspeed")
```

Categorical vs Categorical significance test

Using Chi-square test of independence

```
Null Hypothesis:
    There's no relationship between the two columns

Alternate Hypothesis:
    There's a relationship between the two columns
```

```
8. def chi_square_test(col1, col2):
    data = df[[col1, col2]] # Creating a dataframe from the columns passed
# Converting data to an object of type Table2x2 since stats package is more
compatible with it
    table = sm.stats.Table.from_data(data)
    chi_square_test = table.test_nominal_association() # Running chi-square test
    print("p value:%.4f" % chi_square_test.pvalue)
# Decision rule
    if chi_square_test.pvalue > 0.05:
        print("Accept null hypothesis >> There's no relationship between ", col1,
" and ", col2)
    else:
        print("Reject null hypothesis >> There's a relationship between ", col1,
" and ", col2)

9. # Using the method above to test for significance between columns

chi_square_test("season", "holiday") chi_square_test("season", "workingday")
chi_square_test("season", "weather") chi_square_test("holiday", "workingday")
chi_square_test("holiday", "weather") chi_square_test("workingday", "weather")
```

Numerical vs Categorical significance test¶

Using ANOVA

```
10. def hypothesis_season_and_others(compare_with):
# Each line creates a series for all occurrences of each season type under the
column passed into the functions
    fall = df[df["season"] == "fall"][compare_with]
    spring = df[df["season"] == "spring"][compare_with]
    summer = df[df["season"] == "summer"][compare_with]
    winter = df[df["season"] == "winter"][compare_with]

# Using stats package to run a one-way anova on the data above
    result = stats.f_oneway(winter.values, summer.values, spring.values,
             fall.values)
    print("p value:%.4f" % result.pvalue)

# Decision rule

    if result.pvalue > 0.05:
        print("Accept null hypothesis >> The relationship between season and",
      compare_with, "is not significant")
    else:
        print("Reject null hypothesis >> The relationship between season and",
      compare_with, "is significant")
```

```
# Using function in 10 above to test hypothesis
hypothesis_season_and_others("temp")
hypothesis_season_and_others("temp_feel")
hypothesis_season_and_others("humidity")
hypothesis_season_and_others("windspeed")


11. def hypothesis_weather_and_others(compare_with):

# Each line creates a series for all occurrences of each weather type under the
column passed into the functions
    clear = df[df["weather"] == "Clear or partly cloudy"][compare_with]
    rain = df[df["weather"] == "Light snow or rain"][compare_with]
    mist = df[df["weather"] == "Mist"][compare_with]
    snow = df[df["weather"] == "heavy rain/ice pellets/snow + fog"][compare_with]

# Using stats package to run a one-way anova on the data above
    result = stats.f_oneway(clear.values, rain.values, mist.values,snow.values)
    print("p value:%.4f" % result.pvalue) # print answer in 4 decimal places

# Decision rule
    if result.pvalue > 0.05:
        print("Accept null hypothesis >> The relationship between season and",
compare_with, "is not significant")
    else:
        print("Reject null hypothesis >> The relationship between season and",
compare_with, "is significant")

# Using function in 11 above to test hypothesis
hypothesis_weather_and_others("temp")
hypothesis_weather_and_others("temp_feel")
hypothesis_weather_and_others("humidity")
hypothesis_weather_and_others("windspeed")
```

Using 2-sample t-test

Run descriptive statistics to know if the 2 variables (Yes, No) have equal variance

```
12. df.groupby(["holiday"]).describe()
```

The two variables do not have equal variance under any column. Therefore, we carry out an independent t test with unequal variances

```
13. def hypothesis_holiday_others(compare_with):
# Each line creates a series for all occurrences of Yes and No resp under the
column passed into the functions
    holi_yes_temp = df[df["holiday"] == "Yes"][compare_with]
    holi_no_temp = df[df["holiday"] == "No"][compare_with]

# Running an independent t-test with unequal variances
    result = stats.ttest_ind(holi_no_temp, holi_yes_temp, equal_var= False)
    print("p value:%.4f" % result.pvalue)

# Decision rule
    if result.pvalue > 0.05:
        print("Accept null hypothesis >> The relationship between holiday and",
       compare_with, "is not significant")
    else:
        print("Reject null hypothesis >> The relationship between holiday and",
         compare_with, "is significant")

# using the function above to test for significance
hypothesis_holiday_others("temp")
hypothesis_holiday_others("temp_feel")
hypothesis_holiday_others("humidity")
hypothesis_holiday_others("windspeed")
```

Run descriptive statistics to know if the 2 variables (Yes/No) have equal variance

```
14. df.groupby(["workingday"]).describe()
```

The two variables have equal variance under any column. Therefore, we carry out an independent t test with equal variances

```
15. def hypothesis_workingday_others(compare_with):
# Each line creates a series for all occurrences of Yes and No resp under the
column passed into the functions
    wd_yes_temp = df[df["workingday"] == "Yes"][compare_with]
    wd_no_temp = df[df["workingday"] == "No"][compare_with]

    # Running an independent t-test with equal variances
     result = stats.ttest_ind(wd_no_temp, wd_yes_temp, equal_var= True)
     print("p value:%.4f" % result.pvalue)

# Decision rule
     if result.pvalue > 0.05:
         print("Accept null hypothesis >> The relationship between working day
     and", compare_with, "is not significant")
     else:
         print("Reject null hypothesis >> The relationship between working day
     and", compare_with, "is significant")


# using the function above to test for significance
hypothesis_workingday_others("temp")
hypothesis_workingday_others("temp_feel")
hypothesis_workingday_others("humidity")
hypothesis_workingday_others("windspeed")
```

Appendix J (Cyclical/Seasonality)

```
1. # Import libraries
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from scipy import stats
import statsmodels.api as sm

2. df = pd.read_csv("car_sharing_cleaned",index_col = "timestamp",
parse_dates=True), df.drop("id", axis = 1, inplace = True) # drop id column


3. df_2017 = df.iloc[:5422]

# Creating a template for the plots
fig, ax = plt.subplots(figsize=(15, 6))

# Using ACF plot to determine seasonality
plot_acf(df_2017["temp"],lags=50, title = "Temperature Autocorrelation")
plot_acf(df_2017["humidity"],lags=50, title = "Humidity Autocorrelation")
plot_acf(df_2017["windspeed"],lags=50, title = "Windspeed Autocorrelation")
plot_acf(df_2017["demand"],lags=50, title = "Demand Autocorrelation")

# Time series plot of temperature
4. df_2017["temp"].plot(xlabel = "time", ylabel = "temperature", title =
"Temperature Time series", ax = ax)

5. # Resampling humidity column to daily data and then plotting
df_h_resample = df_2017["humidity"].resample("D").mean().fillna(method = "ffill")
df_h_resample.plot(xlabel = "time", ylabel = "humidity", title = "Humidity Time
series (Daily)", ax = ax)

6. # Resampling windspeed column to daily data and then plotting
df_w_resample = df_2017["windspeed"].resample("D").mean().fillna(method =
"ffill")
df_w_resample.plot(xlabel = "time", ylabel = "windspeed", title = "Windspeed Time
series (Daily)", ax = ax)

7. # Resampling demand column to daily data and then plotting
df_d_resample = df_2017["demand"].resample("D").mean().fillna(method = "ffill")
df_d_resample.plot(xlabel = "time", ylabel = "demand", title = "Demand Time
series (Daily)", ax = ax)
```

Appendix K (ARIMA Model)

```
1. # Import libraries
import pandas as pd
from matplotlib import pyplot as plt
import plotly.express as px
from scipy import stats
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Read csv file into a dataframe and set the timestamp column as the index.
# parse_dates makes pandas read in the datetime columns as datatime columns
2. df = pd.read_csv("car_sharing_cleaned",index_col = "timestamp",
parse_dates=True), df.drop("id", axis = 1, inplace = True) # drop id column


# Resampling df to provide the mean "demand" for each week
# and using forward fill to impute any missing values
3. df_resample = df["demand"].resample("W").mean().fillna(method = "ffill")

# Creates a layout for the figure to be drawn
fig, ax = plt.subplots(figsize = (15,6))

# ACF plot for the data in df_resample
4. plot_acf(df_resample, ax = ax)

# PACF plot for the data in df_resample
5. plot_pacf(df_resample, ax = ax)

# Creating a 30% cutoff
6. cutoff_test = int(len(df_resample) * 0.70)

# Assigning 70% of the data to training set and 30% to the test set
7. y_train = df_resample.iloc[:cutoff_test]
8. y_test = df_resample.iloc[cutoff_test:]
```

```
# Funtion for testing hyperparameters
9. def arima(p,q):
    # Build model
    model = ARIMA(y_train, order = (p,0,q)).fit()
    # Compute MAE
    start = len(y_train)
    end = len(y_train)+len(y_test)-1
    y_predict = model.predict(start=start, end = end)
    mae = mean_absolute_error(y_test, y_predict)
    print("Test MAE for", p, "0", q, "is:", mae)

# Significant values gotten from PACF and ACF plot
10. acf_values = [0,1,2,3,4,5]
11. pacf_values = [0,1,2,11]

# Training a model with every combination of hyperparameters in the lists above
12. for i in acf_values:
     for j in pacf_values:
        arima(j, i)

# The best model is 2,0,3 since it gives the lowest MAE
13. model = ARIMA(y_train, order = (2,0,3)).fit()

start = len(y_train) # Starting point to be used in predict funtion
end = len(y_train)+len(y_test)-1 # End point to be used in predict function

# using model to predict and storing the result
14. y_predict = model.predict(start=start, end = end)

mae = mean_absolute_error(y_test, y_predict) # Calculating MAE
print("Test MAE:", mae)


# A DataFrame with two columns: "y_test" and "y_predict".
# The first contains the true values for the test set, and the second contains
the model's predictions

15. df_pred_test = pd.DataFrame(
    {"y_test": y_test.values, "y_pred": y_predict.values}, index=y_test.index
      )

# Time series plot for the values in the dataframe
fig = px.line(df_pred_test, labels={"value": "demand"},range_y=[0,6], title = "")

fig.show()
```

Appendix L (RFR and Neural Network)

```
1. # Import libraries
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor

2. # Read in data and drop timestamp column
df = pd.read_csv("car_sharing_cleaned",index_col = "id" )

df.drop("timestamp", axis = 1, inplace = True)

3. # Encode categorical variables
df_dummy = pd.get_dummies(df)

4. # Rename columns and drop unnecessary columns
df_dummy.rename(columns={'holiday_Yes':'holiday', "workingday_Yes":
"workingday"}, inplace=True)
df_dummy.drop(["holiday_No", "workingday_No"],axis = 1, inplace=True)

5. # Split data into features and target column
target = "demand"
y = df_dummy[target]
X = df_dummy.drop("demand", axis = 1)

6. # Split data into training and test sets. Using 20% of the data for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2,
random_state= 42)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

```
7. # Building and fitting the model of the training dataset
model = RandomForestRegressor(n_estimators= 200, random_state=42).fit(X_train,
y_train)

# Using the model to predict values for y
8. y_pred_training = model.predict(X_test)

# Calculating the Mean Squared Erroe
9. mse_training = mean_squared_error(y_test,y_pred_training)

10. print("Training MSE:", mse_training)


11. # Initialize neural network model
      model = MLPRegressor(random_state=42, max_iter=500).fit(X_train, y_train)

12. y_pred_training = model.predict(X_test)

13. mse_training = mean_squared_error(y_test,y_pred_training)
14. print("Training MSE:", mse_training)
```

Appendix M (Classifiers)

```
1. # Import libraries
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier

2. # Read CSV file into dataframe
df = pd.read_csv("car_sharing_cleaned",index_col = "id" )
df.drop("timestamp", axis = 1, inplace = True)

3. # Creating binary target column
avg_demand = round(df["demand"].mean(), 6) # Calculating the average demand
# Creating a new binary column that assigns 0 for low demands and 1 for high
demands
4. df["high_demand"] = (df["demand"] > avg_demand).astype(int)

# Changing “0” for low demand to “2”, as per the question
5. df["high_demand"].replace(0, 2, inplace = True)

. # Encode categorical variables
6. df_dummy = pd.get_dummies(df)

7. # Rename columns and drop unnecessary columns
df_dummy.rename(columns={'holiday_Yes':'holiday', "workingday_Yes":
"workingday"}, inplace=True)
df_dummy.drop(["holiday_No", "workingday_No"],axis = 1, inplace=True)

8. # Splitting data
target = "high_demand"
X = df_dummy.drop(columns = target)
y = df_dummy[target]

9. # Splitting data into train and test set. Using 30% of the data for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.3,
random_state= 42)
```

Build a Logistic Regression Classifier

```
# Build and fit the model
model = LogisticRegression(max_iter= 1000).fit(X_train, y_train)

# Test the accuracy of the model
10. lr_acc = model.score(X_test, y_test)

print("Test Accuracy:", round(lr_acc, 2))
```

Build Decision Tree Classifier

```
# Build and fit the model
dt_clf = DecisionTreeClassifier(max_depth=5).fit(X_train, y_train)

# Test the accuracy of the model
11. dt_acc = dt_clf.score(X_test,y_test)
print("Test Accuracy:", round(dt_acc, 2))
```

Build Gradient Boosting Classifier

```
# Build and fit the model
gb_clf = GradientBoostingClassifier().fit(X_train, y_train)

# Test the accuracy of the model
12. gb_acc = gb_clf.score(X_test,y_test)
print("Test Accuracy:", round(gb_acc, 2))
```

Appendix N (Clustering)

```
1. # Import libraries
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler

2. # Read columns needed from the CSV file
df = pd.read_csv("car_sharing_cleaned",usecols=["id", "temp"])
df.head()

3. # Split the dataset to get data in 2017
df_2017 = df.iloc[:5422]

4. # Scalar to scale the values in the dataset
scalar = MinMaxScaler()

5. # Scale columns
df_2017["id_use"] = scalar.fit_transform(df_2017[["id"]])

df_2017["temp_use"] = scalar.fit_transform(df_2017[["temp"]])

6. # Assign data to be used to variable X
X = df_2017
```

```
7. # Create a KMeans clustering function
def kmeans_cluster(n):
    # Build model
    model = KMeans(n_clusters= n, random_state= 42)
    # Fit model to data
    model.fit(X)
    labels = model.labels_
    # Predict data with model
    y_kmeans = model.predict(X)
    print("For,", n, "clusters, value count is:\n",
pd.DataFrame(y_kmeans).value_counts())


8. # Use function created above to cluster data
kmeans_cluster(2)
kmeans_cluster(3)
kmeans_cluster(4)
kmeans_cluster(12)


9. # Create a Gaussian clustering function
def gaussian_cluster(n):
    # Build model
    model = GaussianMixture(n_components= n, n_init = 5, random_state= 42)
    # Fit model to data
    model.fit(X)

    y_kmeans = model.predict(X)
    # Check the distribution
    print("For,", n, "clusters, value count is:\n", pd.Data-
Frame(y_kmeans).value_counts())

10. # Use function created above to cluster data
gaussian_cluster(2)
gaussian_cluster(3)
gaussian_cluster(4)
gaussian_cluster(12)
```