
# Data Analytics and Databases

## CSC-40054-2023-SEM2-A

**School of Computing and Mathematics**

**Keele University, UK**

**2024**

Date of submission: March 29, 2024

Student name: Jyothesh Karnam

Student ID: 23032448

Module Leader: Wenjuan Zhou

## PART 1 DATABASE MANAGEMENT

### 1)

| | id | timestamp | season | holiday | workingday | weather | temp | temp_feel | humidity | windspeed | demand |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 01/01/2017 00:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 81.000000 | 0.000000 | 2.772589 |
| **1** | 2 | 01/01/2017 01:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.688879 |
| **2** | 3 | 01/01/2017 02:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.465736 |
| **3** | 4 | 01/01/2017 03:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 2.564949 |
| **4** | 5 | 01/01/2017 04:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 0.000000 |

- **Database Connection and Creating Table:** The code starts by connecting to the SQLite database and names it CarSharing.db. It creates a table named CarSharing using ‘CREATE TABLE’ statement with columns and also drops if there is an existing table with the same name.
- **Importing Data:** From a CSV file named CarSharing, the code imports all the data into the CarSharing table using ‘to_sql’ method in pandas.
- **Backup Table:** Uses the ‘CREATE TABLE CarSharingBackup AS SELECT * FROM CarSharing’ to create a backup table which has the same data as the original.
- **Database Management and Output:** The code commits all the changes made and closes the connection to the database. Next it displays the data formatted in pandas.

### 2)

| | id | timestamp | season | holiday | workingday | weather | temp | temp_feel | humidity | windspeed | demand | temp_category |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 01/01/2017 00:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 81.000000 | 0.000000 | 2.772589 | Mild |
| **1** | 2 | 01/01/2017 01:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.688879 | Mild |
| **2** | 3 | 01/01/2017 02:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.465736 | Mild |
| **3** | 4 | 01/01/2017 03:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 2.564949 | Mild |
| **4** | 5 | 01/01/2017 04:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 0.000000 | Mild |

- **Adding Column:** The code adds a column named ‘temp_category’ to the CarSharing table using ALTER TABLE and ADD COLUMN statement.

- **Data Categorization:** The code updates the ‘temp_category’ column with the values based on ‘temp_feel’ column using ‘UPDATE’ query with a ‘CASE’ expression. The values in ‘temp_feel’ are evaluated to sort temperatures into cold, mild and hot categories.
- **Database Management and Output:** The code commits all the changes made in the table an and displays the data formatted in pandas.

**3)**

Temperature Data:

| | temp | temp_feel | temp_category |
|---|---|---|---|
| **0** | 9.840000 | 14.395000 | Mild |
| **1** | 9.020000 | 13.635000 | Mild |
| **2** | 9.020000 | 13.635000 | Mild |
| **3** | 9.840000 | 14.395000 | Mild |
| **4** | 9.840000 | 14.395000 | Mild |

Updated CarSharing Data:

| | id | timestamp | season | holiday | workingday | weather | humidity | windspeed | demand | temp_category |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2017-01-01 00:00:00 | spring | No | No | Clear or partly cloudy | 81.000000 | 0.000000 | 2.772589 | Mild |
| **1** | 2 | 2017-01-01 01:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.688879 | Mild |
| **2** | 3 | 2017-01-01 02:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.465736 | Mild |
| **3** | 4 | 2017-01-01 03:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 2.564949 | Mild |
| **4** | 5 | 2017-01-01 04:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 0.000000 | Mild |

- **Creating Table:** Using the ‘CREATE TABLE’ SQL command the code creates a new table named temperature with columns namely ‘temp’, ‘temp_feel’ and ‘temp_category’.
- **Inserting Data:** Using the ‘INSERT INTO SELECT’ SQL command, the code uses the corresponding data from the CarSharing table and inserts the data into the temperature table.
- **Modifying CarSharing Table:** The code creates a temporary table named ‘CarSharing_temp’ and uses ‘SELECT’ statement to add all the columns excluding ‘temp’ and ‘temp_feel’. Then it drops the original CarSharing table and renames the ‘CarSharing_temp’ table into ‘CarSharing’ table using ‘ALTER TABLE RENAME TO’ SQL command.
- **Database Management and Output:** Changes made are then committed and the results are formatted in pandas and shown as output.

**4)**

| | id | timestamp | season | holiday | workingday | weather | humidity | windspeed | demand | temp_category | weather_code |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2017-01-01 00:00:00 | spring | No | No | Clear or partly cloudy | 81.000000 | 0.000000 | 2.772589 | Mild | 1 |
| **1** | 2 | 2017-01-01 01:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.688879 | Mild | 1 |
| **2** | 3 | 2017-01-01 02:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.465736 | Mild | 1 |
| **3** | 4 | 2017-01-01 03:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 2.564949 | Mild | 1 |
| **4** | 5 | 2017-01-01 04:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 0.000000 | Mild | 1 |

- **Calculating Weather Conditions:** The code creates a temporary table named ‘WeatherCodes’ and using the ‘ROW_NUMBER()’ function it assigns the unique numerical codes to distinct weather conditions.
- **Adding Column:** Next the code adds a column named ‘weather_code’ into the ‘CarSharing’ table using ‘ALTER TABLE’ and ‘ADD COLUMN’ statement.
- **Updating Weather Codes:** Using an ‘UPDATE’ query, the code inserts the data into the ‘weather_code’ by matching the conditions in temporary ‘WeatherCodes’ table.
- **Database Management and Output:** After updating the table, it commits the changes made and displays the output formatted with pandas.

**5)**

| | weather_code | weather |
|---|---|---|
| **0** | 1 | Clear or partly cloudy |
| **1** | 2 | Mist |
| **2** | 3 | Light snow or rain |
| **3** | 4 | heavy rain/ice pellets/snow + fog |

| | id | timestamp | season | holiday | workingday | humidity | windspeed | demand | temp_category | weather_code |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2017-01-01 00:00:00 | spring | No | No | 81.000000 | 0.000000 | 2.772589 | Mild | 1 |
| **1** | 2 | 2017-01-01 01:00:00 | spring | No | No | 80.000000 | 0.000000 | 3.688879 | Mild | 1 |
| **2** | 3 | 2017-01-01 02:00:00 | spring | No | No | 80.000000 | 0.000000 | 3.465736 | Mild | 1 |
| **3** | 4 | 2017-01-01 03:00:00 | spring | No | No | 75.000000 | 0.000000 | 2.564949 | Mild | 1 |
| **4** | 5 | 2017-01-01 04:00:00 | spring | No | No | 75.000000 | 0.000000 | 0.000000 | Mild | 1 |

- **Creating Table:** Uses ‘CREATE TABLE IF NOT EXISTS’ statement and creates a table named ‘weather’ with columns ‘weather_code’ and ‘weather’

- **Inserting Data:** Uses ‘INSERT OR IGNORE INTO’ command and inserts the data from ‘weather’ and ‘weather_code’ columns into newly created ‘weather’ table.
- **Modifying CarSharing Table:** Next it creates a ‘CarSharing_temp’ table and uses ‘SELECT’ command to select all the columns and excludes the ‘weather’ column and drops the original ‘CarSharing’ table. Next it uses ‘ALTER TABLE RENAME TO’ command to rename the ‘CarSharing_temp’ table into ‘CarSharing’ table.
- **Database Management and Output:** After the changes are made. The code commits the changes and displays the output styled with pandas.

**6)**

| | timestamp | hour | weekday | month |
|---|---|---|---|---|
| **0** | 2017-01-02 00:00:00 | 0 | Monday | January |
| **1** | 2017-01-02 01:00:00 | 1 | Monday | January |
| **2** | 2017-01-02 02:00:00 | 2 | Monday | January |
| **3** | 2017-01-02 03:00:00 | 3 | Monday | January |
| **4** | 2017-01-02 04:00:00 | 4 | Monday | January |

- **Creating Table:** the code creates a table named ‘time’ using the ‘CREATE TABLE’ statement with columns namely ‘timestamp’, ‘hour’, ‘weekday’ and ‘month’.
- **Inserting Data:** The code utilizes ‘INSERT INTO’ and ‘SELECT’ statements to insert the data into ‘time’ table from CarSharing table. For timestamp it uses strftime() to transform it into hour, weekday and month and also uses ‘CASE’ statement to convert numerical dates into text. Also excludes the weekends.
- **Database Management and Output:** Once the changes are made. the updates are committed and the output is displayed styled with pandas.

**7)**

**7a)**

Date and Time with Highest Demand Rate in 2017 (Excluding Weekends):

| timestamp | max_demand |
|---|---|
| 2017-06-15 17:00:00 | 6.458338 |

The code creates a table named 'HighestDemand2017' and creates two columns named 'timestamp' and 'max_demand'. Using SQL queries, it fetches the data of the timestamp with the highest demand omitting the weekends. The data is then inserted into the 'HighestDemand2017' table. Then the code commits the changes and displays the output which is formatted with pandas.

**7b)**

Weekday, month, and season with the highest and lowest average demand rates in 2017:

| | weekday | month | season | avg_demand | demand_type |
|---|---|---|---|---|---|
| **0** | Monday | June | summer | 4.853999 | Highest |
| **59** | Monday | January | spring | 3.050786 | Lowest |

Within the 'CarSharing.db', the code analyses the 2017 demand data and pinpoints the days with highest and lowest demand rate. It creates a table named 'highest_lowest_avg_demand' to store the results. To calculate the average demands across weekdays, months and seasons the code uses SQL queries such as 'strftime' and 'CASE' statements to filter and categorize the 2017 data. The findings are then inserted into the table and then the changes are committed in the database. Next the results are displayed which is formatted using pandas.

**7c)**

The code firstly analyses hourly average demand rates on the weekday with the highest demand in 2017. Monday being the day identified ad the highest demand, the code then creates a table for storing the results and names the table as 'hourly_demand_selected_weekday'. Then by using SQL query 'SELECT' and 'strftime', it will calculate the average demands by hour. This query will group data by hour finalizes the results in a descending order. Next

the changes are committed and the results are displayed formatted with pandas.

Average demand rate at different hours for Monday throughout 2017:

| | hour | avg_demand |
|---|---|---|
| 0 | 13 | 5.643554 |
| 1 | 12 | 5.621972 |
| 2 | 14 | 5.554613 |
| 3 | 15 | 5.515115 |
| 4 | 16 | 5.503753 |
| 5 | 11 | 5.437365 |
| 6 | 17 | 5.399252 |
| 7 | 10 | 5.223831 |
| 8 | 18 | 5.215943 |
| 9 | 19 | 4.990500 |
| 10 | 20 | 4.726986 |
| 11 | 09 | 4.638345 |
| 12 | 21 | 4.464856 |
| 13 | 00 | 4.230482 |
| 14 | 22 | 4.188675 |
| 15 | 01 | 3.976929 |
| 16 | 08 | 3.934944 |
| 17 | 23 | 3.799622 |
| 18 | 02 | 3.768969 |
| 19 | 03 | 3.074058 |
| 20 | 07 | 3.007593 |
| 21 | 06 | 2.002182 |
| 22 | 05 | 1.743429 |
| 23 | 04 | 1.659888 |

**7d)**

**Subtask d.1:**

```
Most prevalent temperature category in 2017 on weekdays: Mild (Occurrences: 1769)
```

The code identifies the most prevalent temperature category in 2017 by SQL query that selects the ‘temp_category’ and counts the occurrences in 2017 and excludes the weekends. Next it groups the results by ‘temp_category’ and then based on count, it picks the top result using ‘LIMIT 1’ statement.

**Subtask d.2:**

```
Most prevalent weather condition in 2017 on weekdays: Clear or partly cloudy (Occurrences: 2520)
```

This code finds the most prevalent weather condition in 2017 on weekdays through ‘SELECT’, ‘COUNT’, ‘GROUP’ statements. Skips weekends and counts each weather conditions occurrences in 2017 and groups the result and finds the most frequent one.

**Subtask d.3:**

Wind Speed Analysis for Each Month in 2017 (Excluding Weekends):

| | Month | Average Wind Speed | Maximum Wind Speed | Minimum Wind Speed |
|---|---|---|---|---|
| 0 | January | 14.476016 | 39.000700 | 0.000000 |
| 1 | February | 15.636455 | 51.998700 | 0.000000 |
| 2 | March | 15.597622 | 40.997300 | 0.000000 |
| 3 | April | 16.042593 | 40.997300 | 0.000000 |
| 4 | May | 12.615892 | 40.997300 | 0.000000 |
| 5 | June | 12.528143 | 35.000800 | 0.000000 |
| 6 | July | 12.406818 | 56.996900 | 0.000000 |
| 7 | August | 12.574349 | 43.000600 | 0.000000 |
| 8 | September | 11.957809 | 40.997300 | 0.000000 |
| 9 | October | 10.753541 | 32.997500 | 0.000000 |
| 10 | November | 10.670451 | 32.997500 | 0.000000 |
| 11 | December | 10.584651 | 43.000600 | 0.000000 |

The code uses SQL queries like ‘SELECT’, ‘CASE’, ‘AVG’, ‘MAX’, ‘MIN’, ‘strftime’ that analyses average, highest and lowest windspeed for each month and excludes the weekends. The code arranges the data in a chronological manner. Thus, the table shows the insights on how windspeed had varied throughout the year on weekdays.

**Subtask d.4:**

Humidity Analysis for Each Month in 2017 (Excluding Weekends):

| | Month | Average Humidity | Maximum Humidity | Minimum Humidity |
|---|---|---|---|---|
| 0 | January | 55.826923 | 100.000000 | 28.000000 |
| 1 | February | 53.147541 | 100.000000 | 15.000000 |
| 2 | March | 54.524752 | 100.000000 | 0.000000 |
| 3 | April | 61.376206 | 100.000000 | 22.000000 |
| 4 | May | 70.955432 | 100.000000 | 24.000000 |
| 5 | June | 58.538710 | 94.000000 | 20.000000 |
| 6 | July | 62.110390 | 94.000000 | 33.000000 |
| 7 | August | 61.137313 | 94.000000 | 25.000000 |
| 8 | September | 75.495146 | 100.000000 | 48.000000 |
| 9 | October | 73.289552 | 100.000000 | 35.000000 |
| 10 | November | 70.240385 | 100.000000 | 27.000000 |
| 11 | December | 68.335484 | 100.000000 | 26.000000 |

This code analysis the average, maximum and minimum humidity for each month on 2017 and excludes weekends. The code uses SQL queries such as ‘SELECT’, ‘CASE’, ‘AVG’, ‘MAX’, ‘MIN’, ‘strftime’ and provides a breakdown of each month’s humidity levels. This shows how the humidity levels have changed over the months.

**Subtask d.5:**

Average Demand Rate for Each Temperature Category in 2017:

| | Temperature Category | Average Demand |
|---|---|---|
| **0** | Hot | 4.730872 |
| **1** | Mild | 3.963292 |
| **2** | Cold | 3.235517 |

This code calculates average demand rate for each temperature category in 2017 by excluding weekends by using SQL query such as ‘SELECT’, ‘QUERY’, ‘ORDER’. It results by grouping temperature category and sorting the data in descending order which is based on average demand.

**7e)**

Information for July (Month with the Highest Demand):

| Month | Average Demand | Average Wind Speed | Maximum Wind Speed | Minimum Wind Speed | Average Humidity | Maximum Humidity | Minimum Humidity |
|---|---|---|---|---|---|---|---|
| **July** | 4.787655 | 12.015846 | 56.996900 | 0.000000 | 60.292035 | 94.000000 | 17.000000 |

Information Summary for All Months in 2017:

| Month | Average Demand | Average Wind Speed | Maximum Wind Speed | Minimum Wind Speed | Average Humidity | Maximum Humidity | Minimum Humidity |
|---|---|---|---|---|---|---|---|
| **April** | 4.049236 | 15.852275 | 40.997300 | 0.000000 | 66.248899 | 100.000000 | 22.000000 |
| **August** | 4.642341 | 12.411122 | 43.000600 | 0.000000 | 62.173626 | 94.000000 | 25.000000 |
| **December** | 4.276869 | 10.836460 | 43.000600 | 0.000000 | 65.180617 | 100.000000 | 26.000000 |
| **February** | 3.679483 | 15.577717 | 51.998700 | 0.000000 | 53.580717 | 100.000000 | 8.000000 |
| **January** | 3.388312 | 13.748052 | 39.000700 | 0.000000 | 56.307692 | 100.000000 | 28.000000 |
| **July** | 4.787655 | 12.015846 | 56.996900 | 0.000000 | 60.292035 | 94.000000 | 17.000000 |
| **June** | 4.723880 | 11.827618 | 35.000800 | 0.000000 | 58.370861 | 100.000000 | 20.000000 |
| **March** | 3.745415 | 15.974884 | 40.997300 | 0.000000 | 55.997753 | 100.000000 | 0.000000 |
| **May** | 4.571585 | 12.427391 | 40.997300 | 0.000000 | 71.371429 | 100.000000 | 24.000000 |
| **November** | 4.439538 | 12.142271 | 36.997400 | 0.000000 | 64.169231 | 100.000000 | 27.000000 |
| **October** | 4.562444 | 10.892052 | 36.997400 | 0.000000 | 71.571429 | 100.000000 | 29.000000 |
| **September** | 4.550090 | 11.564080 | 40.997300 | 0.000000 | 74.840355 | 100.000000 | 42.000000 |

This code calculates the 2017's demand data and monthly weather in the database. The code analysis and pinpoints the highest average demand by using SQL commands. First the code creates a table named 'info_summary_demand2017' with columns namely average demand, average windspeed, average humidity, maximum windspeed, minimum windspeed and maximum and minimum humidity. To identify the month with peak demand, 'SELECT month FROM info_summary_demand2017 ORDER BY avg_demand DESC LIMIT 1' SQL commands are used. Next the other queries in the code such as 'SELECT * FROM infor_summary_demand2017 WHERE month =?' are used to retrieve important information and organize summary for all months. The changes made are committed and saved into the database and the results are then displayed and formatted with pandas.

## PART 2 DATA ANALYTICS

**1)**

```
Loading the CSV file...
CSV file loaded.
Dropping duplicate rows...
Dropped 0 duplicate rows.
Handling null values...
Filled null values in 'temp' with mean.
Filled null values in 'temp_feel' with mean.
Filled null values in 'humidity' with mean.
Filled null values in 'windspeed' with mean.
Normalizing numerical columns excluding 'id'...
Normalization of numerical columns completed, excluding 'id'.
Preprocessing complete. Data saved to 'CarSharingDataAnalytics.csv'.
```

**Loading CSV File:** The code uses 'pandas.read_csv(filepath)' to load a specific CSV file path into a dataframe.

**Data Cleaning:**

- Removing Duplicate Rows:
  Removing duplicate rows is an important step to achieve accurate results. It ensures that all the data in the dataset are unique. The method used to drop the duplicate rows in the dataset is ‘pandas.DataFrame.drop_duplicates()’.
- Handling Null Values:
  Numerical Data: In order to fill the null values with numerical data the code uses mean method. This method calculates the average value of the series excluding the null values and imputes the missing values with the average value by using ‘fillna(dataframe[column].mean())’ method.
  Categorical Data: To fill the null values in categorical columns the code uses mode method. This technique will use the most frequent occurring values and fills the null values by using ‘fillna(dataframe[column].mode())’ method.

**Data Preprocessing:**

- Data Normalization:
  The code uses ‘MinMaxScalar’ tool from ‘sklearn.preprocessing’ module to normalize the numerical data in a uniform range between 0 and 1. The code excludes the id column and categorical data while performing normalization.

**Exporting Processed Data:**

Next the code will save all the changes made with data cleaning and data preprocessing techniques and exports the changes into a new CSV file named ‘CarSharingDataAnalytics.csv’.

**2)**

```
ANOVA results for season:
                sum_sq      df           F        PR(>F)
C(season)    20.715197     3.0  150.064822  8.024922e-95
Residual    400.504865  8704.0         NaN           NaN

ANOVA results for holiday:
                sum_sq      df         F    PR(>F)
C(holiday)    0.000535     1.0  0.011054  0.916267
Residual    421.219527  8706.0       NaN       NaN

ANOVA results for workingday:
                  sum_sq      df         F    PR(>F)
C(workingday)    0.138734     1.0  2.868367  0.090372
Residual       421.081328  8706.0       NaN       NaN

ANOVA results for weather:
                sum_sq      df          F        PR(>F)
C(weather)    6.937639     3.0  48.586185  3.927930e-31
Residual    414.282423  8704.0        NaN           NaN

                            OLS Regression Results
==============================================================================
Dep. Variable:                 demand   R-squared:                       0.252
Model:                            OLS   Adj. R-squared:                  0.251
Method:                 Least Squares   F-statistic:                     732.1
Date:                Mon, 25 Mar 2024   Prob (F-statistic):               0.00
Time:                        13:56:19   Log-Likelihood:                2094.2
No. Observations:                8708   AIC:                            -4178.
Df Residuals:                    8703   BIC:                            -4143.
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.6212      0.011     58.581      0.000       0.600       0.642
temp           0.0263      0.028      0.950      0.342      -0.028       0.081
temp_feel      0.4053      0.026     15.312      0.000       0.353       0.457
humidity      -0.3314      0.011    -30.262      0.000      -0.353      -0.310
windspeed      0.0759      0.015      5.069      0.000       0.047       0.105
==============================================================================
Omnibus:                      935.459   Durbin-Watson:                   0.310
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1266.177
Skew:                          -0.886   Prob(JB):                    1.13e-275
Kurtosis:                       3.590   Cond. No.                         25.6
==============================================================================

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

**Data Preparation:**

- Loading Dataset:
  The code uses pandas to load the dataset including numerical and categorial columns for analysis.
- Classifying Columns:
  Now, based on the data types, the code will classify the columns into numerical and categorial for hypothesis testing and excludes the timestamp column.

**Hypothesis Testing for Numerical Columns:**

Pearson Correlation method: In order to measure the linear relationship between each numerical column, the code utilizes the Pearson's correlation coefficient testing.

- Null Hypothesis (H0): Proposes that there is no relationship between the column and demand. But this is a default assumption.
- Alternative Hypothesis (H1): Proposes that there is a significant relationship between the column and demand.
- Interpreting Results: p-value is used to summarize each tests outcome. To consider Null hypothesis or not, a threshold value of p-value < 0.05 is used. If the p-value is less than the threshold, then it is strong evidence to conclude that there is a significant relationship between column and demand.

**Hypothesis Testing for Categorical Columns:**

ANOVA Test: In order to check if mean demand is significantly different from other categorial values, the ANOVA (Analysis of Variance) method is used. ANOVA uses three or more samples to compare the means and understand if at least one of them is different from other samples.

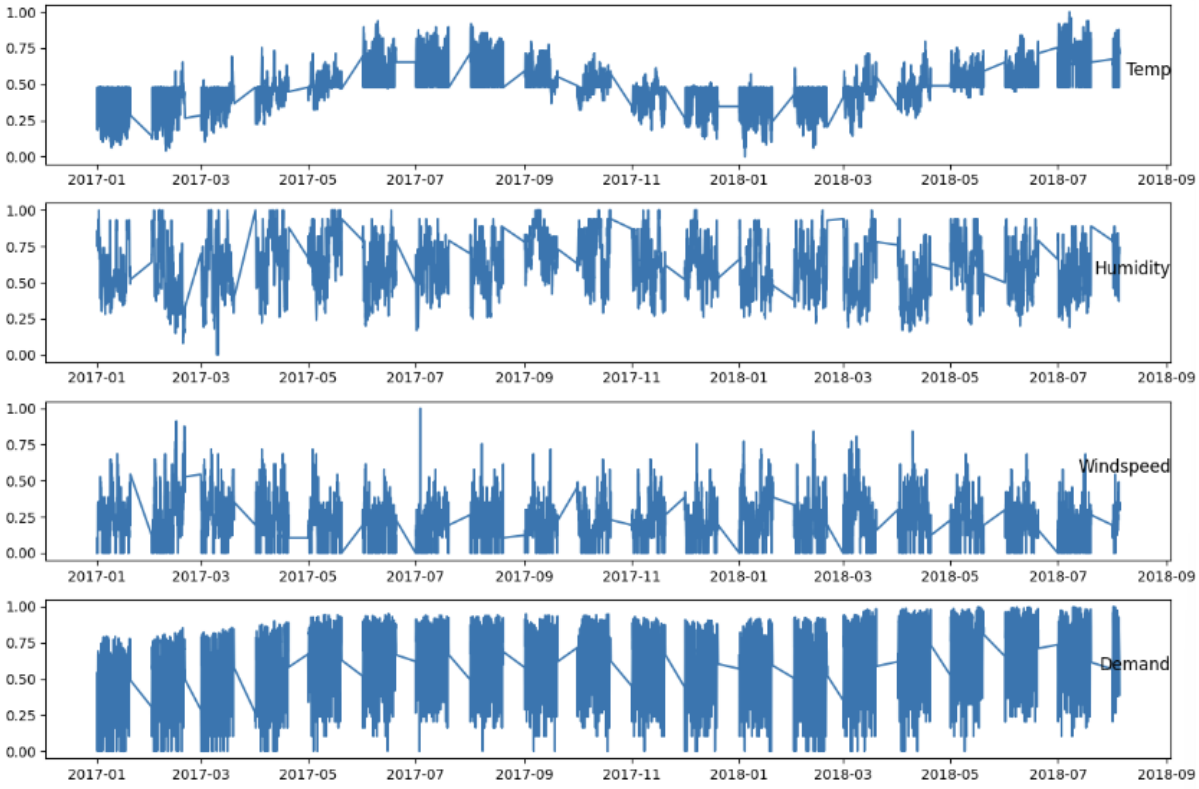
- Model fitting: Utilizes the 'ols' function from 'statsmodels.formula.api' module to fit the linear model. The 'ols' function develops a linear relationship between demand and a categorical column.
- ANOVA Execution: After fitting the linear model, based on the models predictions, ANOVA tests the fitted OLS model with 'statsmodels' library to check if there is any variance in demand across different categories of the independent variable.

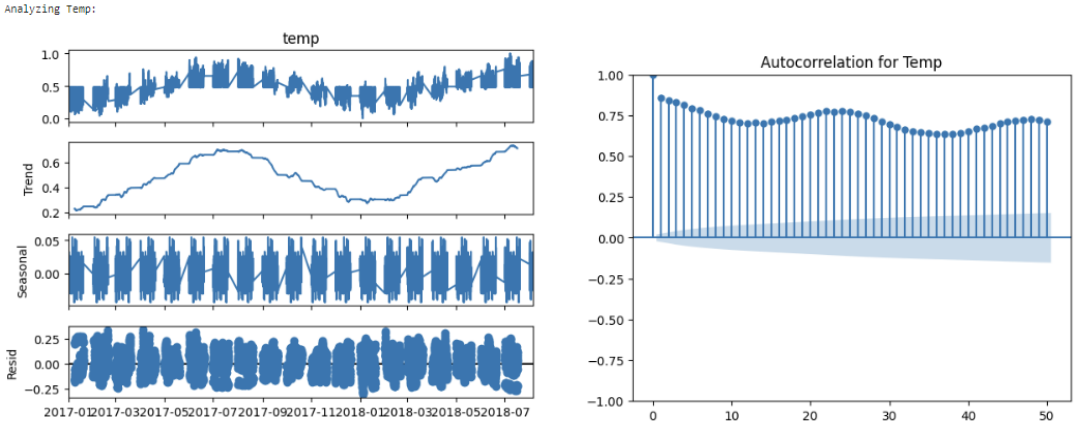
**Results:**

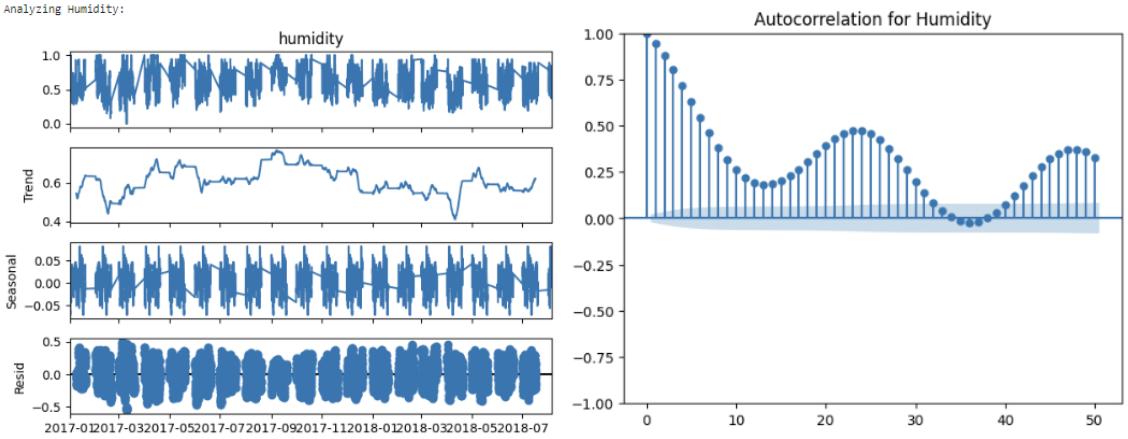
After obtaining p-values from executing ANOVA on the OLS model, it prints the results of p-values and correlation statistics for numerical columns and p-values for categorical columns.

**3)**

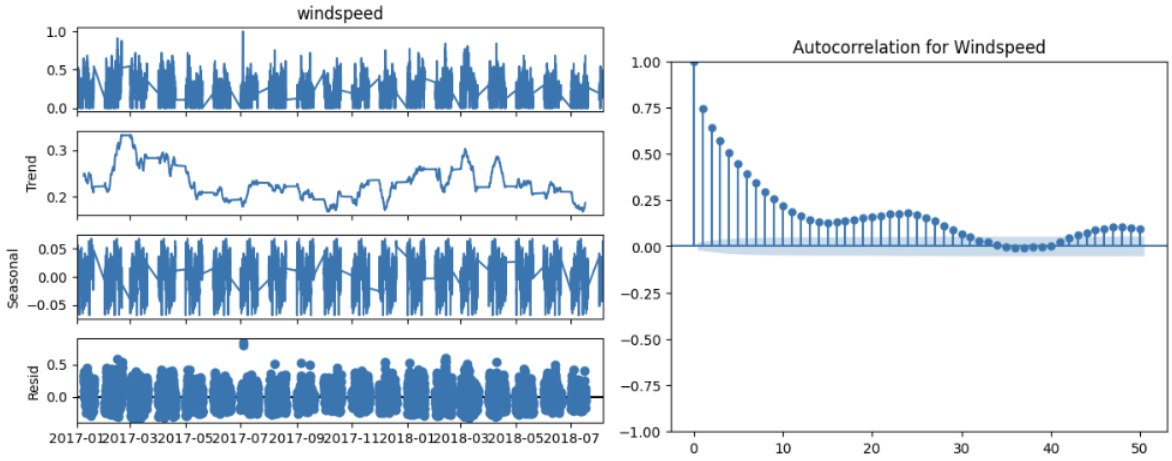
**Time Series Analysis**


Temp
Humidity
Windspeed
Demand
2017-01
2017-03
2017-05
2017-07
2017-09
2017-11
2018-01
2018-03
2018-05
2018-07
2018-09

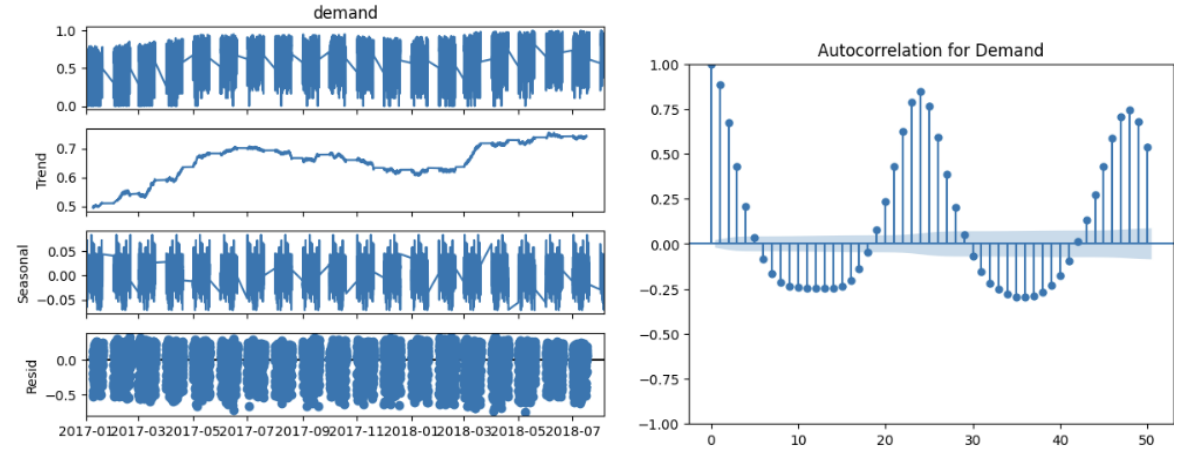

Analyzing Temp:
temp
Trend
Seasonal
Resid
Autocorrelation for Temp


Analyzing Humidity:
humidity
Trend
Seasonal
Resid
Autocorrelation for Humidity

```
Analyzing Windspeed:
```




```
Analyzing Demand:
```




```
Summary of Findings:
1. Temperature shows a clear seasonal pattern, with higher values in summer and lower in winter.
2. Humidity does not display a strong seasonal pattern but varies more randomly over the year.
3. Windspeed shows some seasonality, with higher speeds observed in certain months.
4. Demand appears to have a strong seasonal component, peaking in certain seasons and lower in others.
5. Autocorrelation plots reveal significant correlations at specific lags for temperature and demand, indicating seasonality.
6. The decomposition plots for each variable help to visually separate the trend, seasonal, and residual components, providing insights into
the underlying patterns in the data.
```

**Data Preparation:**

The pre-processed dataframe is loaded from the ‘CarSharingDataAnalytics.csv’ file and for time series analysis, the column named ‘timestamp’ is converted into datatime format and set to DataFrame index.

**Time Series Analysis:**

A time series plot is generated for each variable namely temp, humidity, windspeed and demand. This visual analysis will help reveal trends and patterns in 2017 data.

**Seasonal Decomposition and Autocorrelation Analysis:**

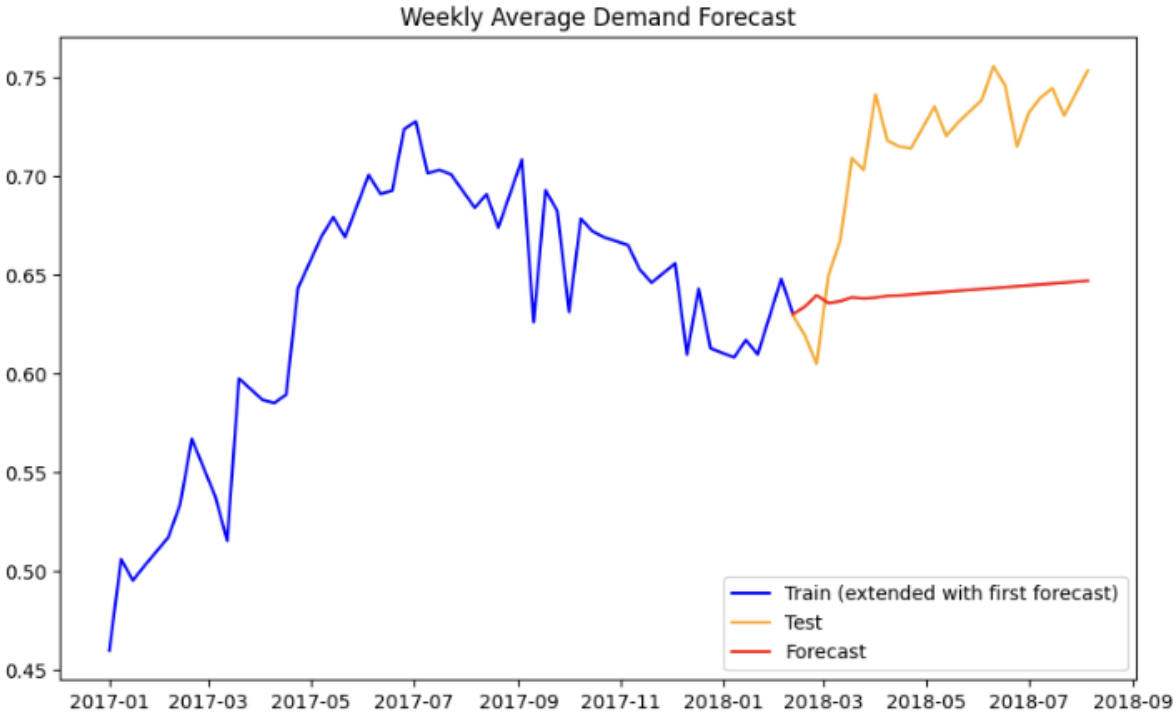
- Seasonal Decomposition: In order to decompose each variables time series into seasonal, trend and residual components, it utilizes ‘seasonal_decompose’ function from ‘statsmodels.tsa.seasonal’ module for decomposition. Parameters such as ‘model=additive’ and ‘period=365’ are used by assuming the daily data and additive model to attain yearly seasonality. The ‘model=additive’ parameter is used to specify that the time series is composed of components that are added together linearly. The ‘period=365’ parameter is used to identify yearly seasonal effects accurately by indicating the cycle length and daily data over an year in this case.
- Autocorrelation Analysis: In order to visualize autocorrelations, the code uses ‘plot_acf’ from ‘statsmodels.graphics.tsaplots’ module to visualize up to 50 lags. Positive significant spikes in autocorrelation plot indicates a constant pattern at specific intervals. This analysis is important for model selection for time series analysis. This visual tool is important to recognize the repeating patterns over intervals within the data.

**Summary of Analysis:**

- The Analysis shows a seasonal pattern in variables and demand. Reveals the changing impacts on humidity and windspeed. Also updates the datasets

**4)**

```
ADF Statistic: -1.9528720170329528
p-value: 0.3076114019162724
Performing stepwise search to minimize aic
 ARIMA(0,2,0)(0,0,0)[0] intercept   : AIC=-189.355, Time=0.03 sec
 ARIMA(1,2,0)(0,0,0)[0] intercept   : AIC=-210.705, Time=0.02 sec
 ARIMA(0,2,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.06 sec
 ARIMA(0,2,0)(0,0,0)[0]             : AIC=-191.348, Time=0.02 sec
 ARIMA(2,2,0)(0,0,0)[0] intercept   : AIC=-232.400, Time=0.02 sec
 ARIMA(3,2,0)(0,0,0)[0] intercept   : AIC=-236.261, Time=0.09 sec
 ARIMA(4,2,0)(0,0,0)[0] intercept   : AIC=-241.860, Time=0.07 sec
 ARIMA(5,2,0)(0,0,0)[0] intercept   : AIC=-246.505, Time=0.11 sec
 ARIMA(5,2,1)(0,0,0)[0] intercept   : AIC=-247.295, Time=0.25 sec
 ARIMA(4,2,1)(0,0,0)[0] intercept   : AIC=-248.644, Time=0.17 sec
 ARIMA(3,2,1)(0,0,0)[0] intercept   : AIC=-249.485, Time=0.16 sec
 ARIMA(2,2,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.12 sec
 ARIMA(3,2,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.19 sec
 ARIMA(2,2,2)(0,0,0)[0] intercept   : AIC=-247.013, Time=0.19 sec
 ARIMA(4,2,2)(0,0,0)[0] intercept   : AIC=-245.858, Time=0.22 sec
 ARIMA(3,2,1)(0,0,0)[0]             : AIC=-250.403, Time=0.11 sec
 ARIMA(2,2,1)(0,0,0)[0]             : AIC=-252.184, Time=0.06 sec
 ARIMA(1,2,1)(0,0,0)[0]             : AIC=-246.107, Time=0.08 sec
 ARIMA(2,2,0)(0,0,0)[0]             : AIC=-234.399, Time=0.01 sec
 ARIMA(2,2,2)(0,0,0)[0]             : AIC=-251.279, Time=0.14 sec
 ARIMA(1,2,0)(0,0,0)[0]             : AIC=-212.703, Time=0.01 sec
 ARIMA(1,2,2)(0,0,0)[0]             : AIC=-251.405, Time=0.13 sec
 ARIMA(3,2,0)(0,0,0)[0]             : AIC=-238.260, Time=0.06 sec
 ARIMA(3,2,2)(0,0,0)[0]             : AIC=-248.210, Time=0.12 sec

Best model:  ARIMA(2,2,1)(0,0,0)[0]
Total fit time: 2.448 seconds
Recommended ARIMA Order: (2, 2, 1)
Test RMSE: 0.0805041154106365
```




**Data Preparation:**

The code loads the dataset and for accurate forecasting, the code handles NaN values in ‘demand’ column using interpolate() function to ensure data completeness.

**Weekly Aggregation:**

The code utilizes ‘pandas.DataFrame.resample(‘W’).mean()’ to aggregate daily demand data into weekly averages. This process provides a clearer view of demand trends over time and allows to analyse demand data on a weekly basis.

**Stationarity and Model Selection:**

Time series forecasting models predict demand by applying mathematical methods derived from previous data and due to this we can surely utilise historical data to estimate future demand because it is predicated on the idea that the future is an extension of the past (Fatah, J. *et al.*, 2018).

- ADF Test: to verify the stationarity, the code uses Augmented Dickey-Fuller test on a training dataset. This step provides key assumption for ARIMA models.
- Auto ARIMA: To determine the optimal ARIMA model parameters, the code utilizes the ‘auto_arima’ function. This function simplifies the model specification process.

**Forecasting and Evaluation:**

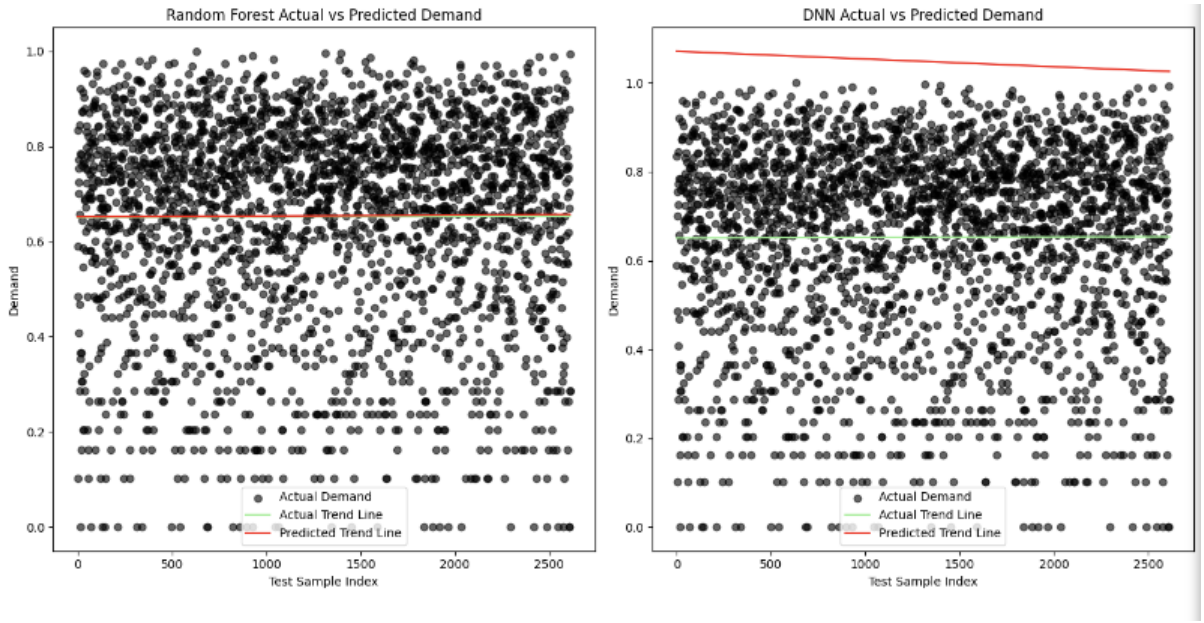
- Fitting ARIMA model: To train the dataset, the code fits the ARIMA model to find the underlying demand patterns.
- Forecasting: By applying the trained ARIMA model, the code generates demand forecast for the test period.
- Performance Metrics: In order to provide the measure of prediction metrics, the code evaluates forecast accuracy through the Root Mean Squared Error (RMSE).

**Visualization:**

The code shows the direct visual comparison between historical training data, the forecasted values generated by ARIMA model and the actual outcomes in the testing period. this visualisation helps identify how well the model shows underlying demand patterns and trends.

**5)**

```
Random Forest MSE: 0.0024
82/82 ━━━━━━━━━━━━━━━━━━━━ 0s 772us/step
DNN MSE: 1.0046
Random Forest performs better with a lower MSE.

Sample Actual vs. Predicted Values (Random Forest):
     Actual  RF_Predicted
0  0.589978      0.559834
1  0.834746      0.872222
2  0.102048      0.152573
3  0.485228      0.467939
4  0.754373      0.791741

Sample Actual vs. Predicted Values (DNN):
     Actual  DNN_Predicted
0  0.589978       1.990257
1  0.834746       1.153160
2  0.102048      -0.454384
3  0.485228       0.764976
4  0.754373      -0.747933
```




**Data Preparation:**

Uses pre-processed dataset and generates features and target variables. From features it will exclude the demand column. Uses ‘float32’ to convert features for DNN compatibility. It also splits the data in a 70:30 ratio for training and testing sets.

**Model Training:**

Random Forest Regressor:

- A flexible machine learning algorithm that produces accurate results even without tuning as it is highly sensitive to the training data. It is a collection of multiple random decision trees.
- A random forest Regressor model with 100 trees is created and trained on the training data.
- Demand rate on the test data is predicted by the model and also calculates the Mean Squared Error.

Deep Neural Network:

- A neural network’s collection of layers, each of which may contain several neurons, is used to describe the transformation between inputs and outputs
- Two hidden layers of DNN are constructed and each of them have ReLU activation and 10 neurons.
- The model is compiled with MSE loss function and Adam optimizer.
- To prevent overfitting, it includes an EarlyStopping callback function and trains on scaled training data.
- MSE is calculated after predictions on test data are made.

**Results and Visualisation:**

- Random forest models MSE and DNN models MSE are compared and the code determines which performs better with a lower MSE value.
- To show better insights of model performance, the code prints sample actual values vs predicted values and also plots the graph that shows the difference between actual and predicted values as a trend line to see which is the best fit.

**6)**

```
Loading the dataset...
Dataset loaded successfully.
Preprocessing data and creating labels...
Labels created. Average demand: 0.66. Label 1 count: 5367, Label 0 count: 3341
Splitting the dataset into training and testing sets...
Dataset split completed.

Training the Decision Tree model...
Decision Tree model trained successfully. Accuracy: 0.9059

Training the Random Forest model...
Random Forest model trained successfully. Accuracy: 0.9162

Training the SVM model...
SVM model trained successfully. Accuracy: 0.6705

Best performing model: Random Forest with an accuracy of 0.9162.
```

**Data Preparation:**

The dataset is loaded into dataframe and the Data is Split into two sections. One is features with pre-processed features except for demand and another one is target ‘demand’. Uses 70% of the data for training and 30% of the data for testing.

**Model Training and Evaluation:**

- Decision Tree Classifier: in order to classify data points, Decision Trees are used which is a Machine Learning algorithm that divide data by asking yes or no question based on features, beginning at a root node.
- Random Forest Regressor: The Random Forest algorithm selects a subset of characteristics for each decision tree, usually the square root or log of the total features and appropriate methods to select dataset subsets.
- Support Vector Machine classifier: Finding a hyperplane in an n-dimensional space that divides the data points into possible classes is the goal. The hyperplane ought to be positioned as far away from the data points as possible.

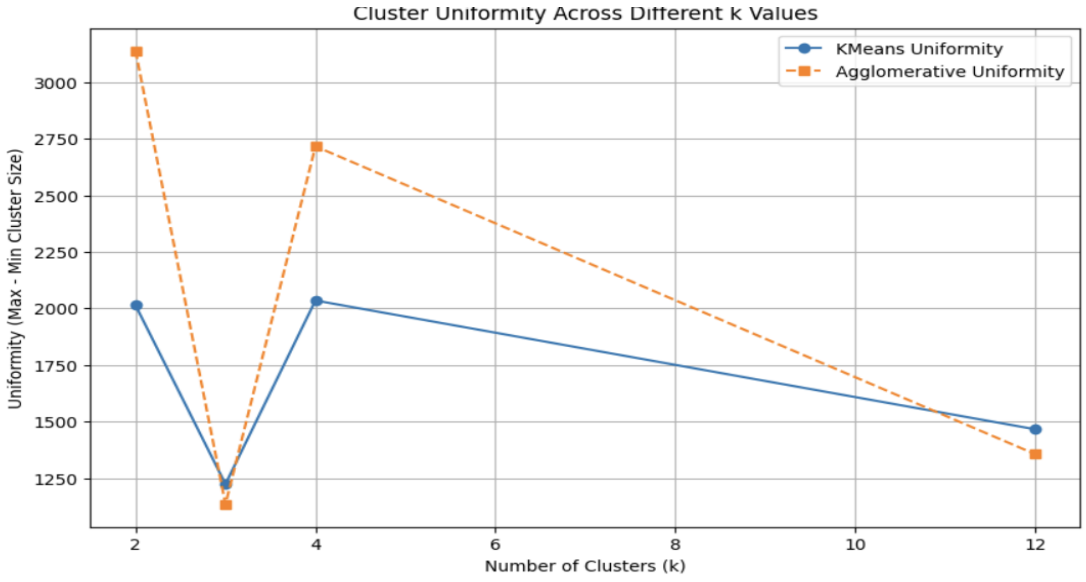
All the three classifiers are trained on the training set and the performance of the classifiers is evaluated using accuracy as a metric on the test set.

**Result:**

The model which has the highest accuracy is identified. Now it is possible to show which classifier is most useful for predicting demand rate categories.

## 7)

```
Loading the dataset...
Dataset loaded successfully.
KMeans Clustering Results:

k = 2
Cluster distribution: {0: 3347, 1: 5361}
Most uniform cluster size difference: 2014

k = 3
Cluster distribution: {0: 2666, 1: 3633, 2: 2409}
Most uniform cluster size difference: 1224

k = 4
Cluster distribution: {0: 2394, 1: 1761, 2: 1259, 3: 3294}
Most uniform cluster size difference: 2035

k = 12
Cluster distribution: {0: 590, 1: 909, 2: 1120, 3: 947, 4: 699, 5: 992, 6: 433, 7: 1558, 8: 334, 9: 92, 10: 603, 11: 431}
Most uniform cluster size difference: 1466


 Agglomerative Clustering Results:

 k = 2
 Cluster distribution: {0: 5923, 1: 2785}
 Most uniform cluster size difference: 3138

 k = 3
 Cluster distribution: {0: 2785, 1: 2394, 2: 3529}
 Most uniform cluster size difference: 1135

 k = 4
 Cluster distribution: {0: 2394, 1: 1972, 2: 3529, 3: 813}
 Most uniform cluster size difference: 2716

 k = 12
 Cluster distribution: {0: 525, 1: 859, 2: 915, 3: 1010, 4: 752, 5: 611, 6: 811, 7: 567, 8: 1558, 9: 202, 10: 408, 11: 490}
 Most uniform cluster size difference: 1356
```




**Loading Dataset:**

The code loads the dataset into a datframe from a CSV file in order to focus on the ‘temp’ column for the 2017 year.

**Clustering Setup and Execution:**

To test different clustering techniques, the code defines a set of k values like 2,3,4 and 12 using KMeans and Agglomerative Clustering methods. The code performs clustering on temperature data for each k value. Hence this method leads to creating clusters based on temperature similarity.

**Evaluation of Uniformity:**

After clustering process, the code evaluates the uniformity score for each of the k by finding the difference in sizes for largest cluster and smallest cluster size. The one **with the lowest score indicates the clusters uniformity.**

**Results and Visualization:**

For each of the k value and clustering method, the code prints out the uniformity and cluster distributions. This helps to identify which k value has the most uniform distribution. Additionally, the code also plots uniformity scores for both KMeans and Agglomerative clustering methods. This will help to see which k value has the most balanced clustering distribution.

## Appendix

**Part 1 Database Management**

**1)**

```
# Import necessary libraries
import sqlite3
import pandas as pd

# Establish database connection
conn = sqlite3.connect('CarSharing.db')

# Step 1: Drop the existing CarSharing table if it exists
drop_table_query = "DROP TABLE IF EXISTS CarSharing;"
conn.execute(drop_table_query)

# Step 2: Create table schema
create_table_query = """
CREATE TABLE CarSharing (
    id INTEGER PRIMARY KEY,
    timestamp TEXT,
    season INTEGER,
    holiday INTEGER,
    workingday INTEGER,
    weather INTEGER,
    temp REAL,
    temp_feel REAL,
    humidity REAL,
    windspeed REAL,
    demand INTEGER
);"""
conn.execute(create_table_query)
```

```
# Step 3: Import data into the created table named "CarSharing"
csv_file_path = r'C:\Users\Jyothesh karnam\Desktop\dadb\CarSharing.csv'
car_data = pd.read_csv(csv_file_path)  # Load data from CSV into a Data-Frame
car_data.to_sql('CarSharing', conn, if_exists='replace', index=False)  # Write DataFrame to SQLite table

# Step 4: Check if backup table exists, drop it if it does
cursor = conn.cursor()
cursor.execute("DROP TABLE IF EXISTS CarSharingBackup;")
cursor.close()

# Create a backup table and copy data
backup_query = "CREATE TABLE CarSharingBackup AS SELECT * FROM CarSharing;"
conn.execute(backup_query)

# Commit changes
conn.commit()

# Fetch the table into a pandas DataFrame
query = "SELECT * FROM CarSharing;"
carsharing_data = pd.read_sql(query, conn)

# Convert timestamp column to datetime format and format it to display both date and time
carsharing_data['timestamp'] = pd.to_datetime(carsharing_data['timestamp']).dt.strftime('%d/%m/%Y %H:%M')

# Close the database connection
conn.close()
```

```
# Apply the center alignment style to each individual cell in the DataFrame
styled_data = carsharing_data.head().style.set_properties(**{'text-align': 'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])

# Display the styled DataFrame
styled_data
```

**Output:**

| | id | timestamp | season | holiday | workingday | weather | temp | temp_feel | humidity | windspeed | demand |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 01/01/2017 00:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 81.000000 | 0.000000 | 2.772589 |
| **1** | 2 | 01/01/2017 01:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.688879 |
| **2** | 3 | 01/01/2017 02:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.465736 |
| **3** | 4 | 01/01/2017 03:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 2.564949 |
| **4** | 5 | 01/01/2017 04:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 0.000000 |

**2)**

```
import sqlite3
import pandas as pd

# Establish database connection
conn = sqlite3.connect('CarSharing.db')
cur = conn.cursor()  # Create a cursor object

# Add a new column temp_category to the CarSharing table with a specified maximum length of 3 characters
try:
    cur.execute("ALTER TABLE CarSharing ADD COLUMN temp_category TEXT(3);")
except sqlite3.OperationalError:
    print("Column 'temp_category' already exists.")
```

```
# Update the values in the temp_category column using a CASE ex-pression
update_query = '''
UPDATE CarSharing
SET temp_category = (
    CASE
        WHEN temp_feel < 10 THEN 'Cold'
        WHEN temp_feel BETWEEN 10 AND 25 THEN 'Mild'
        ELSE 'Hot'
    END
);
'''
cur.execute(update_query)

# Commit the changes to the database
conn.commit()

# Close the cursor|
cur.close()

query = "SELECT * FROM CarSharing;"
data = pd.read_sql(query, conn)
# Close the connection after fetching the data
conn.close()

# Convert timestamp column to datetime format and format it to display both date and time
# data['timestamp'] = pd.to_datetime(data['timestamp']).dt.strftime('%d/%m/%Y %H:%M')

# Apply the center alignment style to each individual cell in the DataFrame
```

```
styled_data = data.head().style.set_properties(**{'text-align':
'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])

# Display the styled DataFrame
styled_data
```

**Output:**

| | id | timestamp | season | holiday | workingday | weather | temp | temp_feel | humidity | windspeed | demand | temp_category |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2017-01-01 00:00:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 81.000000 | 0.000000 | 2.772589 | Mild |
| **1** | 2 | 2017-01-01 01:00:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.688879 | Mild |
| **2** | 3 | 2017-01-01 02:00:00 | spring | No | No | Clear or partly cloudy | 9.020000 | 13.635000 | 80.000000 | 0.000000 | 3.465736 | Mild |
| **3** | 4 | 2017-01-01 03:00:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 2.564949 | Mild |
| **4** | 5 | 2017-01-01 04:00:00 | spring | No | No | Clear or partly cloudy | 9.840000 | 14.395000 | 75.000000 | 0.000000 | 0.000000 | Mild |

**3)**

```
import sqlite3
import pandas as pd  # Import Pandas for data manipulation

# Establish a connection to the database
conn = sqlite3.connect('CarSharing.db')
cursor = conn.cursor()

# Ensure the 'CarSharing' table contains the 'temp_category' column
cursor.execute("PRAGMA table_info(CarSharing);")
columns_info = cursor.fetchall()
column_names = [info[1] for info in columns_info]

if 'temp_category' not in column_names:
    # Add 'temp_category' column if it doesn't exist
```

```
    cursor.execute("ALTER TABLE CarSharing ADD COLUMN temp_category
TEXT;")

# Check if the 'temperature' table exists, drop it if it does to start fresh
cursor.execute("DROP TABLE IF EXISTS temperature;")

# Create the 'temperature' table
cursor.execute("""
CREATE TABLE temperature (
    temp REAL,
    temp_feel REAL,
    temp_category TEXT
);
""")

# Insert data into the 'temperature' table from the 'CarSharing' table
cursor.execute("""
INSERT INTO temperature (temp, temp_feel, temp_category)
SELECT temp, temp_feel, temp_category FROM CarSharing;
""")

# Commit the changes to the database
conn.commit()

# Display the contents of the 'temperature' table to verify
cursor.execute("SELECT * FROM temperature LIMIT 5;")
temperature_data = cursor.fetchall()
df_temperature_data = pd.DataFrame(temperature_data, columns=[desc[0] for
desc in cursor.description])

# Print the DataFrame for temperature data with styling
```

```
styled_temperature_data = df_temperature_data.style.set_properties(**{'text-align': 'center'}).set_table_styles([{'selector': 'th', 'props': [('text-align', 'center')]}])

print("Temperature Data:")

display(styled_temperature_data)


# Create a new table excluding 'temp' and 'temp_feel' columns

cursor.execute("""

CREATE TABLE IF NOT EXISTS CarSharing_temp AS

SELECT id, timestamp, season, holiday, workingday, weather,

      humidity, windspeed, demand, temp_category

FROM CarSharing;

""")


# Drop the original CarSharing table

cursor.execute("DROP TABLE CarSharing;")


# Rename the temporary table to CarSharing

cursor.execute("ALTER TABLE CarSharing_temp RENAME TO CarSharing;")


# Commit the changes to the database

conn.commit()


# Display the contents of the updated CarSharing table


cursor.execute("SELECT * FROM CarSharing LIMIT 5;")

updated_carsharing_data = cursor.fetchall()

df_updated_carsharing_data = pd.DataFrame(updated_carsharing_data, columns=[desc[0] for desc in cursor.description])


# Print the DataFrame for updated CarSharing data with styling
```

```
styled_updated_carsharing_data =
df_updated_carsharing_data.style.set_properties(**{'text-align': 'cen-
ter'}).set_table_styles([{'selector': 'th', 'props': [('text-align', 'center')]}])

print("\nUpdated CarSharing Data:")

display(styled_updated_carsharing_data)


# Close the cursor and connection to clean up

cursor.close()

conn.close()
```

**Output:**

Temperature Data:

| | temp | temp_feel | temp_category |
|---|---|---|---|
| **0** | 9.840000 | 14.395000 | Mild |
| **1** | 9.020000 | 13.635000 | Mild |
| **2** | 9.020000 | 13.635000 | Mild |
| **3** | 9.840000 | 14.395000 | Mild |
| **4** | 9.840000 | 14.395000 | Mild |

Updated CarSharing Data:

| | id | timestamp | season | holiday | workingday | weather | humidity | windspeed | demand | temp_category |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 2017-01-01 00:00:00 | spring | No | No | Clear or partly cloudy | 81.000000 | 0.000000 | 2.772589 | Mild |
| **1** | 2 | 2017-01-01 01:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.688879 | Mild |
| **2** | 3 | 2017-01-01 02:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.465736 | Mild |
| **3** | 4 | 2017-01-01 03:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 2.564949 | Mild |
| **4** | 5 | 2017-01-01 04:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 0.000000 | Mild |

**4)**

```
import sqlite3

import pandas as pd
```

```
# Establish database connection
conn = sqlite3.connect('CarSharing.db')

# Step 1: Create a temporary table with unique weather codes
conn.execute("""
CREATE TEMP TABLE IF NOT EXISTS WeatherCodes AS
SELECT DISTINCT
    weather,
    ROW_NUMBER() OVER (ORDER BY weather) AS code
FROM CarSharing;
""")

# Step 2: Add the 'weather_code' column to the 'CarSharing' table if it doesn't exist
try:
    conn.execute("ALTER TABLE CarSharing ADD COLUMN weather_code INTEGER;")
except sqlite3.OperationalError:
    # If the column already exists, this error will be caught
    print("Column 'weather_code' already exists.")

# Step 3: Update the 'weather_code' column with the corresponding code for each weather value
update_query = """
UPDATE CarSharing
SET weather_code = (
    SELECT code
    FROM WeatherCodes
    WHERE CarSharing.weather = WeatherCodes.weather
);
"""
```

```
conn.execute(update_query)

# Commit the changes to the database
conn.commit()

# Optional: Display rows to verify the new 'weather_code' column
query = "SELECT * FROM CarSharing LIMIT 5;"
updated_data = pd.read_sql(query, conn)

conn.close()

# Apply the center alignment style to each individual cell in the DataFrame
styled_updated_data = updated_data.style.set_properties(**{'text-align': 'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])

# Display the styled DataFrame
styled_updated_data
```

**Output:**

```
Column 'weather_code' already exists.
```

| | id | timestamp | season | holiday | workingday | weather | humidity | windspeed | demand | temp_category | weather_code |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2017-01-01 00:00:00 | spring | No | No | Clear or partly cloudy | 81.000000 | 0.000000 | 2.772589 | Mild | 1 |
| 1 | 2 | 2017-01-01 01:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.688879 | Mild | 1 |
| 2 | 3 | 2017-01-01 02:00:00 | spring | No | No | Clear or partly cloudy | 80.000000 | 0.000000 | 3.465736 | Mild | 1 |
| 3 | 4 | 2017-01-01 03:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 2.564949 | Mild | 1 |
| 4 | 5 | 2017-01-01 04:00:00 | spring | No | No | Clear or partly cloudy | 75.000000 | 0.000000 | 0.000000 | Mild | 1 |

**5)**

```
# Import necessary libraries
import sqlite3
import pandas as pd

# Establish database connection
conn = sqlite3.connect('CarSharing.db')

# Step 1: Create the 'weather' table
create_weather_table_query = """
CREATE TABLE IF NOT EXISTS weather (
    weather TEXT,
    weather_code INTEGER PRIMARY KEY
);
"""

conn.execute(create_weather_table_query)

# Step 2: Insert distinct weather and weather_code pairs into the 'weather' table
# Note: Using INSERT OR IGNORE to avoid duplicate insertion errors
insert_weather_data_query = """
INSERT OR IGNORE INTO weather (weather, weather_code)
SELECT DISTINCT weather, weather_code FROM CarSharing;
"""

conn.execute(insert_weather_data_query)

# Commit the changes to the database
conn.commit()

# Display the contents of the 'weather' table to verify
query = "SELECT * FROM weather;"
```

```
weather_data = pd.read_sql(query, conn)

# Apply the center alignment style to each individual cell in the DataFrame
styled_weather_data = weather_data.style.set_properties(**{'text-align':
'center'}).set_table_styles([{
    'selector': 'th',  # Apply to all headers
    'props': [('text-align', 'center')]
}])

# Display the styled DataFrame
display(styled_weather_data)

# Close the connection to proceed with modifications
conn.close()

# Re-establish database connection to proceed with modifications
conn = sqlite3.connect('CarSharing.db')

# Create a temporary table that excludes the 'weather' and 'temp_category'
columns
create_temp_carsharing_query = """
CREATE TABLE IF NOT EXISTS CarSharing_temp AS
SELECT id, timestamp, season, holiday, workingday, humidity,
      windspeed, demand, temp_category, weather_code
FROM CarSharing;
"""
conn.execute(create_temp_carsharing_query)

# Drop the original CarSharing table
conn.execute("DROP TABLE CarSharing;")
```

```
# Rename the temporary table to CarSharing
conn.execute("ALTER TABLE CarSharing_temp RENAME TO CarSharing;")

# Commit the changes to the database
conn.commit()

# Display the updated CarSharing table
print("\nUpdated CarSharing Table:")
updated_carsharing_data = pd.read_sql("SELECT * FROM CarSharing LIMIT 5;", conn)

# Apply the center alignment style to each individual cell in the DataFrame
styled_updated_carsharing_data = updated_carsharing_data.style.set_properties(**{'text-align': 'center'}).set_table_styles([dict(selector='th', props=[('text-align', 'center')])])

# Display the styled DataFrame
display(styled_updated_carsharing_data)

# Close the database connection
conn.close()
```

**Output:**

| | weather_code | weather |
|---|---|---|
| **0** | 1 | Clear or partly cloudy |
| **1** | 2 | Mist |
| **2** | 3 | Light snow or rain |
| **3** | 4 | heavy rain/ice pellets/snow + fog |

Updated CarSharing Table:

| | id | timestamp | season | holiday | workingday | humidity | windspeed | demand | temp_category | weather_code |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2017-01-01 00:00:00 | spring | No | No | 81.000000 | 0.000000 | 2.772589 | Mild | 1 |
| 1 | 2 | 2017-01-01 01:00:00 | spring | No | No | 80.000000 | 0.000000 | 3.688879 | Mild | 1 |
| 2 | 3 | 2017-01-01 02:00:00 | spring | No | No | 80.000000 | 0.000000 | 3.465736 | Mild | 1 |
| 3 | 4 | 2017-01-01 03:00:00 | spring | No | No | 75.000000 | 0.000000 | 2.564949 | Mild | 1 |
| 4 | 5 | 2017-01-01 04:00:00 | spring | No | No | 75.000000 | 0.000000 | 0.000000 | Mild | 1 |

**6)**

```
# Import necessary libraries
import sqlite3
import pandas as pd

# Establish database connection
conn = sqlite3.connect('CarSharing.db')
cur = conn.cursor()

# Drop the existing "time" table to avoid schema mismatch issues. Use with caution!
cur.execute('DROP TABLE IF EXISTS time;')

# Create the table named "time" with the correct columns
cur.execute('''
CREATE TABLE time(
    timestamp TEXT,
    hour INTEGER,
    weekday TEXT,
    month TEXT);
''')

# Insert rows into the table "time" from "CarSharing" table, with transformations
# Exclude weekends (Saturday and Sunday) from the inserted data
cur.execute('''
```

```
INSERT INTO time (timestamp, hour, weekday, month)
SELECT
    strftime('%Y-%m-%d %H:%M:%S', timestamp) as timestamp,
    strftime('%H', timestamp) as hour,
    CASE strftime('%w', timestamp)
        WHEN '1' THEN 'Monday'
        WHEN '2' THEN 'Tuesday'
        WHEN '3' THEN 'Wednesday'
        WHEN '4' THEN 'Thursday'
        WHEN '5' THEN 'Friday'
        ELSE NULL
    END as weekday,
    CASE strftime('%m', timestamp)
        WHEN '01' THEN 'January'
        WHEN '02' THEN 'February'
        WHEN '03' THEN 'March'
        WHEN '04' THEN 'April'
        WHEN '05' THEN 'May'
        WHEN '06' THEN 'June'
        WHEN '07' THEN 'July'
        WHEN '08' THEN 'August'
        WHEN '09' THEN 'September'
        WHEN '10' THEN 'October'
        WHEN '11' THEN 'November'
        WHEN '12' THEN 'December'
    END as month
FROM CarSharing
WHERE strftime('%w', timestamp) NOT IN ('0', '6');
"")

# Commit the changes to the database
```

```
conn.commit()

# Fetch and print the data structure of the "time" table using Pandas
columns_df = pd.read_sql_query("PRAGMA table_info(time);", conn)
print("Table 'time' columns:", ", ".join(columns_df['name'].tolist()))

# Query and display the first 5 rows of the "time" table using Pandas
time_df = pd.read_sql_query("SELECT * FROM time LIMIT 5;", conn)

# Apply center alignment for the content in the cells and headers
time_df_styled = time_df.style.set_properties(**{'text-align':
'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])

# Display the styled DataFrame
print("\nFirst 5 rows of the 'time' table:")
display(time_df_styled)

# Close the database connection
conn.close()
```

**Output:**

```
Table 'time' columns: timestamp, hour, weekday, month

First 5 rows of the 'time' table:
```

| | timestamp | hour | weekday | month |
|---|---|---|---|---|
| **0** | 2017-01-02 00:00:00 | 0 | Monday | January |
| **1** | 2017-01-02 01:00:00 | 1 | Monday | January |
| **2** | 2017-01-02 02:00:00 | 2 | Monday | January |
| **3** | 2017-01-02 03:00:00 | 3 | Monday | January |
| **4** | 2017-01-02 04:00:00 | 4 | Monday | January |

**7)**

**7a)**

```
import sqlite3
import pandas as pd

# Connect to the database
conn = sqlite3.connect('CarSharing.db')
cursor = conn.cursor()

# Step 1: Create a table to store the highest demand data (drop if exists for re-runs)
cursor.execute('''
DROP TABLE IF EXISTS HighestDemand2017;
''')
cursor.execute('''
CREATE TABLE HighestDemand2017 (
    timestamp TEXT,
    max_demand INTEGER
);
''')
```

```
# Your existing query with slight modification for grouping and ordering
query_highest_demand = """
SELECT timestamp, MAX(demand) AS max_demand
FROM CarSharing
WHERE
    strftime('%Y', timestamp) = '2017'
   AND strftime('%w', timestamp) NOT IN ('0', '6')  -- Exclude Sundays and
Saturdays
GROUP BY timestamp
ORDER BY MAX(demand) DESC
LIMIT 1;
"""

# Execute the query and fetch the result
highest_demand_data = pd.read_sql(query_highest_demand, conn)

# Check if there is any data fetched
if not highest_demand_data.empty:
    # Step 2: Insert the fetched data into the new table
    for _, row in highest_demand_data.iterrows():
        cursor.execute('''
        INSERT INTO HighestDemand2017 (timestamp, max_demand)
        VALUES (?, ?);
        ''', (row['timestamp'], row['max_demand']))

    # Step 3: Commit the insert operations to save changes
    conn.commit()

    # Fetch and display the result from the new table
    highest_demand_result = pd.read_sql('SELECT * FROM HighestDemand2017',
conn)
```

```
    # Apply the center alignment style and hide the index
    styled_highest_demand_result = highest_demand_result.style.set_properties(**{'text-align': 'center'}).set_table_styles([{
        'selector': 'th',  # Apply to all headers
        'props': [('text-align', 'center')]
    }]).hide(axis='index')

    print("Date and Time with Highest Demand Rate in 2017 (Excluding Weekends):")
    display(styled_highest_demand_result)
else:
    print("No data found for the specified criteria.")

# Don't forget to close your database connection when done
conn.close()
```

**Output:**

Date and Time with Highest Demand Rate in 2017 (Excluding Weekends):

| timestamp | max_demand |
|---|---|
| 2017-06-15 17:00:00 | 6.458338 |

**7b)**

```
import sqlite3
import pandas as pd

# Connect to the SQLite database
conn = sqlite3.connect("CarSharing.db")
c = conn.cursor()
```

```
# Explicitly drop the table if it exists to ensure a fresh start
c.execute("DROP TABLE IF EXISTS highest_lowest_avg_demand;")

# Create the table with specific column types, including the new demand_type column
c.execute("""
CREATE TABLE highest_lowest_avg_demand (
    weekday TEXT,
    month TEXT,
    season TEXT,
    avg_demand REAL,
    demand_type TEXT
);
""")

# Calculate the average demand for each combination of weekday, month, and season in 2017, excluding weekends
avg_demand_query = """
SELECT
    CASE strftime('%w', timestamp)
        WHEN '1' THEN 'Monday'
        WHEN '2' THEN 'Tuesday'
        WHEN '3' THEN 'Wednesday'
        WHEN '4' THEN 'Thursday'
        WHEN '5' THEN 'Friday'
    END AS weekday,
    CASE strftime('%m', timestamp)
        WHEN '01' THEN 'January'
        WHEN '02' THEN 'February'
        WHEN '03' THEN 'March'
        WHEN '04' THEN 'April'
```

```
        WHEN '05' THEN 'May'
        WHEN '06' THEN 'June'
        WHEN '07' THEN 'July'
        WHEN '08' THEN 'August'
        WHEN '09' THEN 'September'
        WHEN '10' THEN 'October'
        WHEN '11' THEN 'November'
        WHEN '12' THEN 'December'
    END AS month,
    season,
    AVG(demand) AS avg_demand
FROM CarSharing
WHERE
    strftime('%Y', timestamp) = '2017'
    AND strftime('%w', timestamp) NOT IN ('0', '6')
GROUP BY weekday, month, season
ORDER BY avg_demand DESC
"""
c.execute(avg_demand_query)
avg_demand_details_2017 = c.fetchall()

# Convert the query results into a DataFrame
df_avg_demand_details_2017 = pd.DataFrame(avg_demand_details_2017,
columns=['weekday', 'month', 'season', 'avg_demand'])

# Identify the highest and lowest average demand rates
highest_avg_demand =
df_avg_demand_details_2017.iloc[[0]].assign(demand_type='Highest')
lowest_avg_demand = df_avg_demand_details_2017.iloc[[-
1]].assign(demand_type='Lowest')

# Combine the highest and lowest entries into one DataFrame for display
```

```
df_combined = pd.concat([highest_avg_demand, lowest_avg_demand])


# Convert DataFrame to a list of tuples for the executemany insertion, including the new demand_type
data_to_insert = list(df_combined.itertuples(index=False, name=None))
c.executemany("""
INSERT INTO highest_lowest_avg_demand (weekday, month, season, avg_demand, demand_type)
VALUES (?, ?, ?, ?, ?);
""", data_to_insert)


# Commit the transaction to the database
conn.commit()


# Use Pandas styling for display
styled_df_combined = df_combined.style.set_properties(**{'text-align': 'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])


# Display the styled DataFrame
print("Weekday, month, and season with the highest and lowest average demand rates in 2017:")
display(styled_df_combined)


# Close the database connection
conn.close()
```

**Output:**

Weekday, month, and season with the highest and lowest average demand rates in 2017:

| | weekday | month | season | avg_demand | demand_type |
|---|---|---|---|---|---|
| **0** | Monday | June | summer | 4.853999 | Highest |
| **59** | Monday | January | spring | 3.050786 | Lowest |

**7c)**

```
import sqlite3
import pandas as pd

# Assuming 'highest_avg_demand_weekday' is defined from the previous task
highest_avg_demand_weekday = 'Monday'  # Example, replace with actual variable from subtask B

# Connect to the SQLite database
conn = sqlite3.connect("CarSharing.db")
c = conn.cursor()

# Explicitly drop the table if it exists to ensure a fresh start
c.execute("DROP TABLE IF EXISTS hourly_demand_selected_weekday;")

# Create the table with specific column types
c.execute("""
CREATE TABLE hourly_demand_selected_weekday (
    hour TEXT,
    avg_demand REAL
);
""")

# Execute the query to calculate the average demand rate at different hours for the selected highest average demand weekday throughout 2017
```

```
c.execute(f"""
SELECT
    strftime('%H', timestamp) AS hour,
    AVG(demand) AS avg_demand
FROM CarSharing
WHERE
    strftime('%Y', timestamp) = '2017'
    AND strftime('%w', timestamp) = CASE '{highest_avg_demand_weekday}'
        WHEN 'Monday' THEN '1'
        WHEN 'Tuesday' THEN '2'
        WHEN 'Wednesday' THEN '3'
        WHEN 'Thursday' THEN '4'
        WHEN 'Friday' THEN '5'
    END
GROUP BY hour
ORDER BY avg_demand DESC
""")

hourly_demand_2017 = c.fetchall()

# Convert the result to a pandas DataFrame
df_hourly_demand_2017 = pd.DataFrame(hourly_demand_2017, columns=['hour',
'avg_demand'])

# Insert the DataFrame data into the SQLite database table using SQL commands
# Convert DataFrame to a list of tuples for the executemany insertion
data_to_insert = list(df_hourly_demand_2017.itertuples(index=False, name=None))
c.executemany("""
INSERT INTO hourly_demand_selected_weekday (hour, avg_demand)
VALUES (?, ?);
""", data_to_insert)
```

```
# Commit the transaction to the database
conn.commit()


# Use Pandas styling for display
styled_df_hourly_demand_2017 =
df_hourly_demand_2017.style.set_properties(**{'text-align':
'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])


# Display the styled DataFrame
print(f"\nAverage demand rate at different hours for
{highest_avg_demand_weekday} throughout 2017:")
display(styled_df_hourly_demand_2017)


# Close the database connection
conn.close()
```

**Output:**

Average demand rate at different hours for Monday throughout 2017:

| | hour | avg_demand |
|---|---|---|
| 0 | 13 | 5.643554 |
| 1 | 12 | 5.621972 |
| 2 | 14 | 5.554613 |
| 3 | 15 | 5.515115 |
| 4 | 16 | 5.503753 |
| 5 | 11 | 5.437365 |
| 6 | 17 | 5.399252 |
| 7 | 10 | 5.223831 |
| 8 | 18 | 5.215943 |
| 9 | 19 | 4.990500 |
| 10 | 20 | 4.726986 |
| 11 | 09 | 4.638345 |
| 12 | 21 | 4.464856 |
| 13 | 00 | 4.230482 |
| 14 | 22 | 4.188675 |
| 15 | 01 | 3.976929 |
| 16 | 08 | 3.934944 |
| 17 | 23 | 3.799622 |
| 18 | 02 | 3.768969 |
| 19 | 03 | 3.074058 |
| 20 | 07 | 3.007593 |
| 21 | 06 | 2.002182 |
| 22 | 05 | 1.743429 |
| 23 | 04 | 1.659888 |

**7d)**

**Subtask d.1)**

```
import sqlite3


# Reconnect to the SQLite database
conn = sqlite3.connect("CarSharing.db")
c = conn.cursor()


# Query to find the most prevalent temp_category in 2017, excluding Saturdays and Sundays
c.execute("""
SELECT temp_category, COUNT(temp_category) AS count
FROM CarSharing
```

```
WHERE strftime('%Y', timestamp) = '2017'
AND strftime('%w', timestamp) NOT IN ('0', '6') -- Exclude Saturdays and Sundays
GROUP BY temp_category
ORDER BY count DESC
LIMIT 1
""")
most_prevalent_temp_category = c.fetchone()

# Explicitly drop the table if it exists to ensure a fresh start
c.execute("DROP TABLE IF EXISTS MostPrevalentTempCategory;")

# Create the table with specific column types
c.execute("""
CREATE TABLE MostPrevalentTempCategory (
    temp_category TEXT,
    occurrences INTEGER
);
""")

# Insert the result into the newly created table
c.execute("""
INSERT INTO MostPrevalentTempCategory (temp_category, occurrences)
VALUES (?, ?);
""", (most_prevalent_temp_category[0], most_prevalent_temp_category[1]))

# Commit the transaction to the database
conn.commit()

# Close the database connection
conn.close()
```

```
# Print the result
print(f"Most prevalent temperature category in 2017 on weekdays:
{most_prevalent_temp_category[0]} (Occurrences:
{most_prevalent_temp_category[1]})")
```

**Output:**

```
Most prevalent temperature category in 2017 on weekdays: Mild (Occurrences: 1769)
```

**Subtask d.2)**

```
import sqlite3

# Reconnect to the SQLite database
conn = sqlite3.connect("CarSharing.db")
c = conn.cursor()

# Query to find the most prevalent weather condition in 2017, excluding Saturdays
and Sundays
c.execute("""
SELECT w.weather, COUNT(*) AS count
FROM CarSharing cs
JOIN weather w ON cs.weather_code = w.weather_code
WHERE strftime('%Y', cs.timestamp) = '2017'
AND strftime('%w', cs.timestamp) NOT IN ('0', '6')  -- Exclude Saturdays and
Sundays
GROUP BY w.weather
ORDER BY count DESC
LIMIT 1
""")
most_prevalent_weather = c.fetchone()

# Explicitly drop the table if it exists to ensure a fresh start
c.execute("DROP TABLE IF EXISTS MostPrevalentWeather;")
```

```
# Create the table with specific column types
c.execute("""
CREATE TABLE MostPrevalentWeather (
    weather TEXT,
    occurrences INTEGER
);
""")

# Insert the result into the newly created table
c.execute("""
INSERT INTO MostPrevalentWeather (weather, occurrences)
VALUES (?, ?);
""", (most_prevalent_weather[0], most_prevalent_weather[1]))

# Commit the transaction to the database
conn.commit()

# Close the database connection
conn.close()

# Print the result
print(f"Most prevalent weather condition in 2017 on weekdays:
{most_prevalent_weather[0]} (Occurrences: {most_prevalent_weather[1]})")
```

**Output:**

```
Most prevalent weather condition in 2017 on weekdays: Clear or partly cloudy (Occurrences: 2520)
```

**Subtask d.3)**

```
import sqlite3
```

```
import pandas as pd


# Reconnect to the SQLite database
conn = sqlite3.connect("CarSharing.db")
c = conn.cursor()


# Execute the query to analyze wind speed for each month in 2017, excluding Saturdays and Sundays
c.execute("""
SELECT
    CASE strftime('%m', timestamp)
        WHEN '01' THEN 'January'
        WHEN '02' THEN 'February'
        WHEN '03' THEN 'March'
        WHEN '04' THEN 'April'
        WHEN '05' THEN 'May'
        WHEN '06' THEN 'June'
        WHEN '07' THEN 'July'
        WHEN '08' THEN 'August'
        WHEN '09' THEN 'September'
        WHEN '10' THEN 'October'
        WHEN '11' THEN 'November'
        WHEN '12' THEN 'December'
    END AS Month,
    AVG(windspeed) AS avg_wind_speed,
    MAX(windspeed) AS max_wind_speed,
    MIN(windspeed) AS min_wind_speed
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
AND strftime('%w', timestamp) NOT IN ('0', '6')  -- Exclude Saturdays and Sundays
GROUP BY Month
```

```
ORDER BY strftime('%m', timestamp)
""")
wind_speed_stats = c.fetchall()

# Explicitly drop the table if it exists to ensure a fresh start
c.execute("DROP TABLE IF EXISTS WindSpeedStats2017;")

# Create the table with specific column types
c.execute("""
CREATE TABLE WindSpeedStats2017 (
    Month TEXT,
    avg_wind_speed REAL,
    max_wind_speed REAL,
    min_wind_speed REAL
);
""")

# Insert the results into the newly created table
for row in wind_speed_stats:
    c.execute("""
    INSERT INTO WindSpeedStats2017 (Month, avg_wind_speed,
max_wind_speed, min_wind_speed)
    VALUES (?, ?, ?, ?);
    """, row)

# Commit the transaction to the database
conn.commit()

# Close the database connection
conn.close()
```

```
# Convert the results to a pandas DataFrame
df_wind_speed_stats = pd.DataFrame(wind_speed_stats, columns=['Month', 'Average Wind Speed', 'Maximum Wind Speed', 'Minimum Wind Speed'])


# Apply the center alignment style to each individual cell in the DataFrame
styled_df_wind_speed_stats = df_wind_speed_stats.style.set_properties(**{'text-align': 'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])


# Display the styled DataFrame
print("\nWind Speed Analysis for Each Month in 2017 (Excluding Weekends):")
styled_df_wind_speed_stats
```

**Output:**

Wind Speed Analysis for Each Month in 2017 (Excluding Weekends):

| | Month | Average Wind Speed | Maximum Wind Speed | Minimum Wind Speed |
|---|---|---|---|---|
| 0 | January | 14.476016 | 39.000700 | 0.000000 |
| 1 | February | 15.636455 | 51.998700 | 0.000000 |
| 2 | March | 15.597622 | 40.997300 | 0.000000 |
| 3 | April | 16.042593 | 40.997300 | 0.000000 |
| 4 | May | 12.615892 | 40.997300 | 0.000000 |
| 5 | June | 12.528143 | 35.000800 | 0.000000 |
| 6 | July | 12.406818 | 56.996900 | 0.000000 |
| 7 | August | 12.574349 | 43.000600 | 0.000000 |
| 8 | September | 11.957809 | 40.997300 | 0.000000 |
| 9 | October | 10.753541 | 32.997500 | 0.000000 |
| 10 | November | 10.670451 | 32.997500 | 0.000000 |
| 11 | December | 10.584651 | 43.000600 | 0.000000 |

**Subtask d.4)**

```
import sqlite3
import pandas as pd
```

```
# Reconnect to the SQLite database for humidity analysis
conn = sqlite3.connect("CarSharing.db")
c = conn.cursor()


# Execute the query to analyze humidity for each month in 2017, excluding
Saturdays and Sundays
c.execute("""
SELECT
    CASE strftime('%m', timestamp)
        WHEN '01' THEN 'January'
        WHEN '02' THEN 'February'
        WHEN '03' THEN 'March'
        WHEN '04' THEN 'April'
        WHEN '05' THEN 'May'
        WHEN '06' THEN 'June'
        WHEN '07' THEN 'July'
        WHEN '08' THEN 'August'
        WHEN '09' THEN 'September'
        WHEN '10' THEN 'October'
        WHEN '11' THEN 'November'
        WHEN '12' THEN 'December'
    END AS Month,
    AVG(humidity) AS avg_humidity,
    MAX(humidity) AS max_humidity,
    MIN(humidity) AS min_humidity
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
AND strftime('%w', timestamp) NOT IN ('0', '6')  -- Exclude Saturdays and Sundays
GROUP BY Month
ORDER BY strftime('%m', timestamp)
```

```
""")
humidity_stats = c.fetchall()

# Explicitly drop the table if it exists to ensure a fresh start
c.execute("DROP TABLE IF EXISTS HumidityStats2017;")

# Create the table with specific column types
c.execute("""
CREATE TABLE HumidityStats2017 (
    Month TEXT,
    avg_humidity REAL,
    max_humidity REAL,
    min_humidity REAL
);
""")

# Insert the results into the newly created table
for row in humidity_stats:
    c.execute("""
    INSERT INTO HumidityStats2017 (Month, avg_humidity, max_humidity,
min_humidity)
    VALUES (?, ?, ?, ?);
    """, row)

# Commit the transaction to the database
conn.commit()

# Close the database connection
conn.close()

# Convert the results to a pandas DataFrame
```

```
df_humidity_stats = pd.DataFrame(humidity_stats, columns=['Month', 'Average Humidity', 'Maximum Humidity', 'Minimum Humidity'])


# Apply the center alignment style to each individual cell in the DataFrame
styled_df_humidity_stats = df_humidity_stats.style.set_properties(**{'text-align': 'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])


# Display the styled DataFrame
print("\nHumidity Analysis for Each Month in 2017 (Excluding Weekends):")
styled_df_humidity_stats
```

**Output:**

Humidity Analysis for Each Month in 2017 (Excluding Weekends):

| | Month | Average Humidity | Maximum Humidity | Minimum Humidity |
|---|---|---|---|---|
| 0 | January | 55.826923 | 100.000000 | 28.000000 |
| 1 | February | 53.147541 | 100.000000 | 15.000000 |
| 2 | March | 54.524752 | 100.000000 | 0.000000 |
| 3 | April | 61.376206 | 100.000000 | 22.000000 |
| 4 | May | 70.955432 | 100.000000 | 24.000000 |
| 5 | June | 58.538710 | 94.000000 | 20.000000 |
| 6 | July | 62.110390 | 94.000000 | 33.000000 |
| 7 | August | 61.137313 | 94.000000 | 25.000000 |
| 8 | September | 75.495146 | 100.000000 | 48.000000 |
| 9 | October | 73.289552 | 100.000000 | 35.000000 |
| 10 | November | 70.240385 | 100.000000 | 27.000000 |
| 11 | December | 68.335484 | 100.000000 | 26.000000 |

**Subtask d.5)**

```
import sqlite3
import pandas as pd
```

```
# Reconnect to the SQLite database
conn = sqlite3.connect("CarSharing.db")
c = conn.cursor()

# Execute the query to find the average demand rate for each temperature category in 2017 on weekdays
c.execute("""
SELECT
    temp_category,
    AVG(demand) AS avg_demand
FROM CarSharing
WHERE
    strftime('%Y', timestamp) = '2017'
    AND strftime('%w', timestamp) NOT IN ('0', '6')  -- Exclude Saturdays ('6') and Sundays ('0')
GROUP BY temp_category
ORDER BY avg_demand DESC
""")
avg_demand_by_temp_category = c.fetchall()

# Explicitly drop the table if it exists to ensure a fresh start
c.execute("DROP TABLE IF EXISTS AvgDemandByTempCategory;")

# Create the table with specific column types
c.execute("""
CREATE TABLE AvgDemandByTempCategory (
    temp_category TEXT,
    avg_demand REAL
);
""")
```

```
# Insert the results into the newly created table
for row in avg_demand_by_temp_category:
    c.execute("""
    INSERT INTO AvgDemandByTempCategory (temp_category, avg_demand)
    VALUES (?, ?);
    """, row)

# Commit the transaction to the database
conn.commit()

# Close the database connection
conn.close()

# Convert the results to a pandas DataFrame
df_avg_demand_by_temp_category =
pd.DataFrame(avg_demand_by_temp_category, columns=['Temperature Category',
'Average Demand'])

# Apply the center alignment style to each individual cell in the DataFrame
styled_df_avg_demand_by_temp_category =
df_avg_demand_by_temp_category.style.set_properties(**{'text-align':
'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])

# Display the styled DataFrame
print("\nAverage Demand Rate for Each Temperature Category in 2017:")
styled_df_avg_demand_by_temp_category
```

**Output:**

Average Demand Rate for Each Temperature Category in 2017:

| | Temperature Category | Average Demand |
|---|---|---|
| 0 | Hot | 4.730872 |
| 1 | Mild | 3.963292 |
| 2 | Cold | 3.235517 |

**7e)**

```
import sqlite3
import pandas as pd

# Connect to the SQLite database
conn = sqlite3.connect("CarSharing.db")
cur = conn.cursor()

# Drop the existing table to avoid an error if it already exists
cur.execute("DROP TABLE IF EXISTS info_summary_demand2017")

# Create the summary table for all months in 2017 with month names
cur.execute("""
CREATE TABLE info_summary_demand2017 AS
SELECT
    CASE strftime('%m', timestamp)
        WHEN '01' THEN 'January'
        WHEN '02' THEN 'February'
        WHEN '03' THEN 'March'
        WHEN '04' THEN 'April'
        WHEN '05' THEN 'May'
        WHEN '06' THEN 'June'
        WHEN '07' THEN 'July'
        WHEN '08' THEN 'August'
        WHEN '09' THEN 'September'
        WHEN '10' THEN 'October'
```

```
        WHEN '11' THEN 'November'
        WHEN '12' THEN 'December'
    END AS month,
    AVG(demand) AS avg_demand,
    AVG(windspeed) AS avg_windspeed,
    MAX(windspeed) AS max_windspeed,
    MIN(windspeed) AS min_windspeed,
    AVG(humidity) AS avg_humidity,
    MAX(humidity) AS max_humidity,
    MIN(humidity) AS min_humidity
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
GROUP BY month
""")

# Identify the month with the highest average demand in 2017
cur.execute("""
SELECT month
FROM info_summary_demand2017
ORDER BY avg_demand DESC
LIMIT 1
""")
highest_demand_month_name = cur.fetchone()[0]

# Fetch the detailed information for the month with the highest demand
cur.execute("""
SELECT * FROM info_summary_demand2017
WHERE month = ?
""", (highest_demand_month_name,))
highest_month_info = cur.fetchall()
```

```
# Fetch the detailed information for comparison to other months
cur.execute("SELECT * FROM info_summary_demand2017")
all_months_info = cur.fetchall()

# Close the database connection
conn.close()

# Convert the results to pandas DataFrames for better visualization
df_highest_month_info = pd.DataFrame(highest_month_info, columns=['Month', 'Average Demand', 'Average Wind Speed', 'Maximum Wind Speed', 'Minimum Wind Speed', 'Average Humidity', 'Maximum Humidity', 'Minimum Humidity']).set_index('Month')
df_all_months_info = pd.DataFrame(all_months_info, columns=['Month', 'Average Demand', 'Average Wind Speed', 'Maximum Wind Speed', 'Minimum Wind Speed', 'Average Humidity', 'Maximum Humidity', 'Minimum Humidity']).set_index('Month')

# Styling the DataFrames
styled_highest_month_info = df_highest_month_info.style.set_properties(**{'text-align': 'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])
styled_all_months_info = df_all_months_info.style.set_properties(**{'text-align': 'center'}).set_table_styles([{
    'selector': 'th',
    'props': [('text-align', 'center')]
}])

# Display the results with styling
print(f"Information for {highest_demand_month_name} (Month with the Highest Demand):")
display(styled_highest_month_info)
print("\nInformation Summary for All Months in 2017:")
```

display(styled_all_months_info)

**Output:**

Information for July (Month with the Highest Demand):

| Month | Average Demand | Average Wind Speed | Maximum Wind Speed | Minimum Wind Speed | Average Humidity | Maximum Humidity | Minimum Humidity |
|---|---|---|---|---|---|---|---|
| July | 4.787655 | 12.015846 | 56.996900 | 0.000000 | 60.292035 | 94.000000 | 17.000000 |

Information Summary for All Months in 2017:

| Month | Average Demand | Average Wind Speed | Maximum Wind Speed | Minimum Wind Speed | Average Humidity | Maximum Humidity | Minimum Humidity |
|---|---|---|---|---|---|---|---|
| April | 4.049236 | 15.852275 | 40.997300 | 0.000000 | 66.248899 | 100.000000 | 22.000000 |
| August | 4.642341 | 12.411122 | 43.000600 | 0.000000 | 62.173626 | 94.000000 | 25.000000 |
| December | 4.276869 | 10.836460 | 43.000600 | 0.000000 | 65.180617 | 100.000000 | 26.000000 |
| February | 3.679483 | 15.577717 | 51.998700 | 0.000000 | 53.580717 | 100.000000 | 8.000000 |
| January | 3.388312 | 13.748052 | 39.000700 | 0.000000 | 56.307692 | 100.000000 | 28.000000 |
| July | 4.787655 | 12.015846 | 56.996900 | 0.000000 | 60.292035 | 94.000000 | 17.000000 |
| June | 4.723880 | 11.827618 | 35.000800 | 0.000000 | 58.370861 | 100.000000 | 20.000000 |
| March | 3.745415 | 15.974884 | 40.997300 | 0.000000 | 55.997753 | 100.000000 | 0.000000 |
| May | 4.571585 | 12.427391 | 40.997300 | 0.000000 | 71.371429 | 100.000000 | 24.000000 |
| November | 4.439538 | 12.142271 | 36.997400 | 0.000000 | 64.169231 | 100.000000 | 27.000000 |
| October | 4.562444 | 10.892052 | 36.997400 | 0.000000 | 71.571429 | 100.000000 | 29.000000 |
| September | 4.550090 | 11.564080 | 40.997300 | 0.000000 | 74.840355 | 100.000000 | 42.000000 |

**Part 2 Data Analytics**

**1)**

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# Define the CSV file path
csv_file_path = r'C:\Users\Jyothesh karnam\Desktop\dadb\CarSharing.csv'

# Load the CSV file for preprocessing
print("Loading the CSV file...")
```

```
df = pd.read_csv(csv_file_path)
print("CSV file loaded.")

# Drop duplicate rows and report the number dropped
print("Dropping duplicate rows...")
initial_row_count = len(df)
df = df.drop_duplicates()
final_row_count = len(df)
print(f"Dropped {initial_row_count - final_row_count} duplicate rows.")

# Handle null values for numerical and categorical columns
print("Handling null values...")
for column in df.columns:
    if df[column].dtype == 'object' or column == 'id':  # Exclude 'id' and categorical data
        if df[column].isnull().sum() > 0:
            df[column] = df[column].fillna(df[column].mode()[0])
            print(f"Filled null values in '{column}' with mode.")
    else:  # Numeric data
        if df[column].isnull().sum() > 0:
            df[column] = df[column].fillna(df[column].mean())
            print(f"Filled null values in '{column}' with mean.")

# Identify numerical columns to be normalized (exclude 'object' type and 'id')
numerical_columns = df.select_dtypes(exclude=['object']).columns.tolist()
if 'id' in numerical_columns:
    numerical_columns.remove('id')  # Exclude 'id' from normalization

# Initialize MinMaxScaler and normalize numerical columns
print("Normalizing numerical columns excluding 'id'...")
scaler = MinMaxScaler()
```

```
df[numerical_columns] = scaler.fit_transform(df[numerical_columns])
print("Normalization of numerical columns completed, excluding 'id'.")

# Save the preprocessed DataFrame to a new CSV file
preprocessed_csv_file_path = 'CarSharingDataAnalytics.csv'
df.to_csv(preprocessed_csv_file_path, index=False)
print(f"Preprocessing complete. Data saved to '{preprocessed_csv_file_path}'.")
```

**Output:**

```
Loading the CSV file...
CSV file loaded.
Dropping duplicate rows...
Dropped 0 duplicate rows.
Handling null values...
Filled null values in 'temp' with mean.
Filled null values in 'temp_feel' with mean.
Filled null values in 'humidity' with mean.
Filled null values in 'windspeed' with mean.
Normalizing numerical columns excluding 'id'...
Normalization of numerical columns completed, excluding 'id'.
Preprocessing complete. Data saved to 'CarSharingDataAnalytics.csv'.
```

**2)**

```
# Required Libraries
import pandas as pd
import numpy as np
import statsmodels.api as sm
from statsmodels.formula.api import ols
import seaborn as sns
import matplotlib.pyplot as plt

# Load the preprocessed DataFrame
df = pd.read_csv('CarSharingDataAnalytics.csv')

# EDA: Distribution of demand using histplot with red color
```

```
# sns.histplot(df['demand'], kde=True, color='red')

# plt.title('Distribution of Demand')

# plt.show()


# For each continuous variable, using jointplot for scatter in red and regression line in black

for column in ['temp', 'temp_feel', 'humidity', 'windspeed']:  # Adjusted to specific continuous variables

    # Creating a jointplot with scatter in red

    g = sns.jointplot(x=column, y='demand', data=df, kind='scatter', color='red')


    # Adding a regression line in black

    # Note: We fit a separate regression line because seaborn's jointplot doesn't allow different colors

    # for scatter points and regression lines in the same call.

    sns.regplot(x=column, y='demand', data=df, scatter=False, ax=g.ax_joint, color='black')


    plt.close()  # This line will ensure the plot is not shown


# ANOVA for categorical variables

for column in ['season', 'holiday', 'workingday', 'weather']:  # Adjusted to specific categorical variables

    model = ols('demand ~ C({})'.format(column), data=df).fit()

    anova_result = sm.stats.anova_lm(model, typ=2)

    print(f"ANOVA results for {column}:\n", anova_result, "\n")


# OLS Regression for continuous variables

X = df[['temp', 'temp_feel', 'humidity', 'windspeed']]  # Adjusted to specific continuous variables

y = df['demand']


# Adding a constant for intercept
```

```
X = sm.add_constant(X)

# Fit the OLS model
model = sm.OLS(y, X).fit()

# Summary of the model
print(model.summary())

# (If you have any DataFrame modifications, ensure they are done before this step)

# Overwrite the original CSV file with the current state of the DataFrame
df_check = pd.read_csv('CarSharingDataAnalytics.csv')
```

**Output:**

```
ANOVA results for season:
                sum_sq      df           F        PR(>F)
C(season)    20.715197     3.0  150.064822  8.024922e-95
Residual    400.504865  8704.0         NaN           NaN

ANOVA results for holiday:
                sum_sq      df         F    PR(>F)
C(holiday)    0.000535     1.0  0.011054  0.916267
Residual    421.219527  8706.0       NaN       NaN

ANOVA results for workingday:
                  sum_sq      df         F    PR(>F)
C(workingday)    0.138734     1.0  2.868367  0.090372
Residual       421.081328  8706.0       NaN       NaN

ANOVA results for weather:
                sum_sq      df          F        PR(>F)
C(weather)    6.937639     3.0  48.586185  3.927930e-31
Residual    414.282423  8704.0        NaN           NaN

                            OLS Regression Results
==============================================================================
Dep. Variable:                 demand   R-squared:                       0.252
Model:                            OLS   Adj. R-squared:                  0.251
Method:                 Least Squares   F-statistic:                     732.1
Date:                Thu, 28 Mar 2024   Prob (F-statistic):               0.00
Time:                        18:36:29   Log-Likelihood:                2094.2
No. Observations:                8708   AIC:                            -4178.
Df Residuals:                    8703   BIC:                            -4143.
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.6212      0.011     58.581      0.000       0.600       0.642
temp           0.0263      0.028      0.950      0.342      -0.028       0.081
temp_feel      0.4053      0.026     15.312      0.000       0.353       0.457
humidity      -0.3314      0.011    -30.262      0.000      -0.353      -0.310
windspeed      0.0759      0.015      5.069      0.000       0.047       0.105
==============================================================================
Omnibus:                      935.459   Durbin-Watson:                   0.310
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1266.177
Skew:                          -0.886   Prob(JB):                    1.13e-275
Kurtosis:                       3.590   Cond. No.                         25.6
==============================================================================

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

**3)**

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf

# Load the preprocessed DataFrame
df = pd.read_csv('CarSharingDataAnalytics.csv')

# Convert 'timestamp' to datetime and set as index
df['timestamp'] = pd.to_datetime(df['timestamp'])
```

```
df.set_index('timestamp', inplace=True)

# List of variables to analyze
variables = ['temp', 'humidity', 'windspeed', 'demand']

# Visual Analysis: Plotting time series
plt.figure(figsize=(12, 8))
for i, var in enumerate(variables, 1):
    plt.subplot(len(variables), 1, i)
    plt.plot(df[var])
    plt.title(var.capitalize(), y=0.5, loc='right')
plt.tight_layout()
plt.show()

# Decomposition and Autocorrelation Analysis
for var in variables:
    print(f"\nAnalyzing {var.capitalize()}:")

    # Decompose the time series
    decomposition = seasonal_decompose(df[var], model='additive', period=365)
    decomposition.plot()
    plt.show()

    # Autocorrelation plot
    plot_acf(df[var].dropna(), lags=50)
    plt.title(f"Autocorrelation for {var.capitalize()}")
    plt.show()

# Hypothetical Summary of Findings
print("\nSummary of Findings:")
```

print("1. Temperature shows a clear seasonal pattern, with higher values in summer and lower in winter.")

print("2. Humidity does not display a strong seasonal pattern but varies more randomly over the year.")

print("3. Windspeed shows some seasonality, with higher speeds observed in certain months.")

print("4. Demand appears to have a strong seasonal component, peaking in certain seasons and lower in others.")

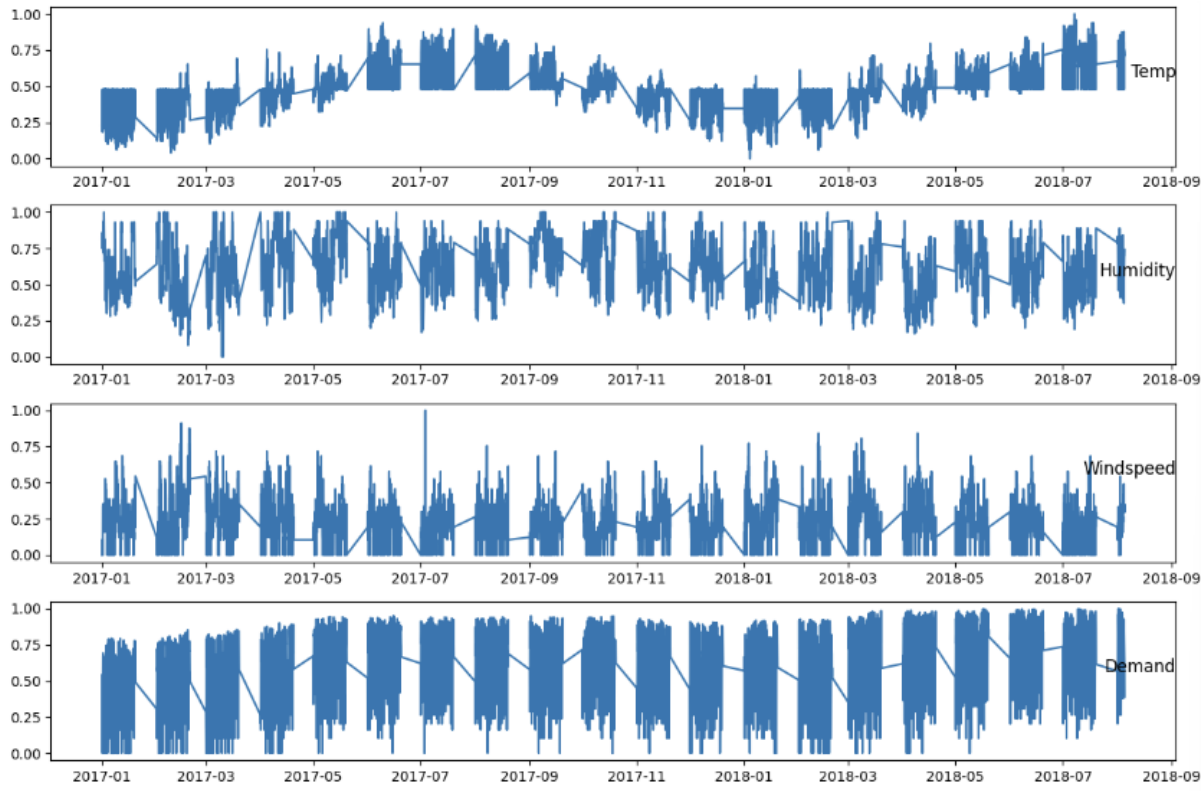
print("5. Autocorrelation plots reveal significant correlations at specific lags for temperature and demand, indicating seasonality.")

print("6. The decomposition plots for each variable help to visually separate the trend, seasonal, and residual components, providing insights into the underlying patterns in the data.")

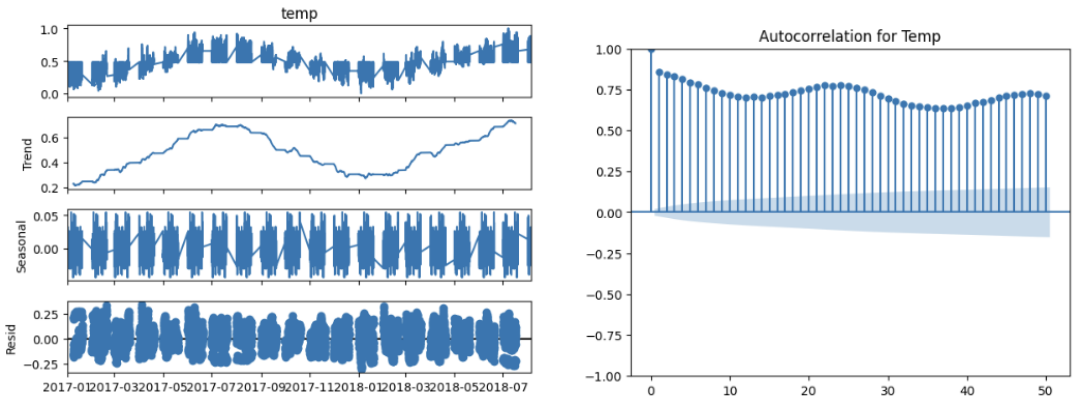
# Overwrite the original CSV file with the current state of the DataFrame (even if unchanged)

df.to_csv('CarSharingDataAnalytics.csv', index=True)

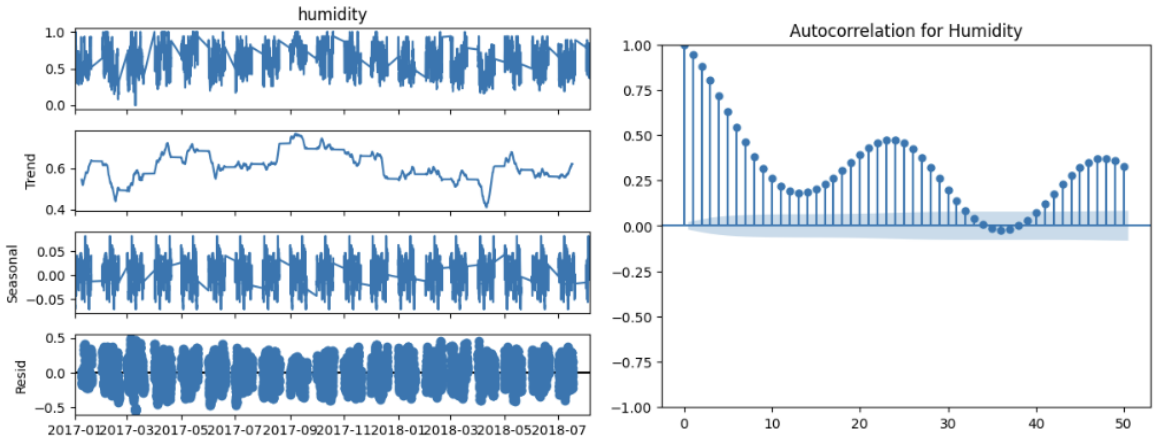
**Output:**



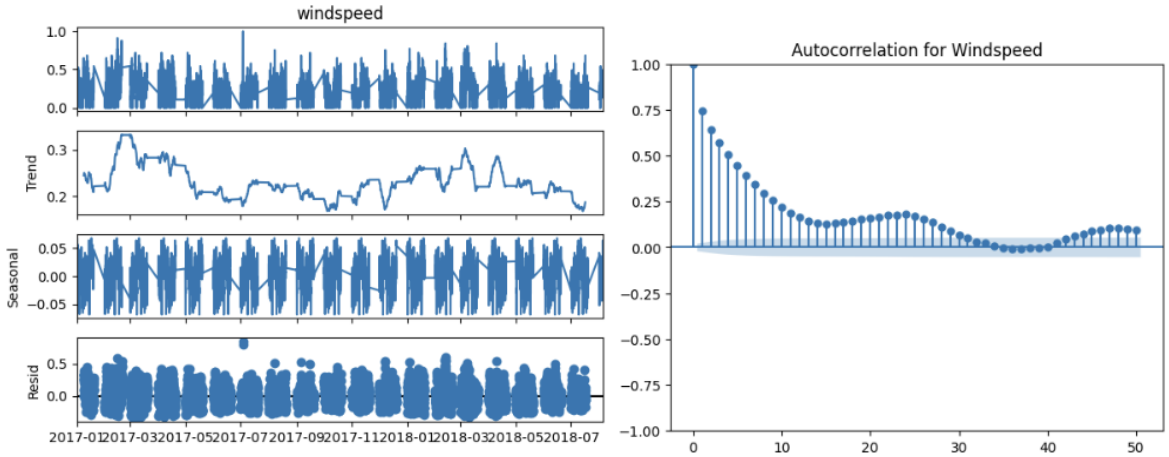
Analyzing Temp:



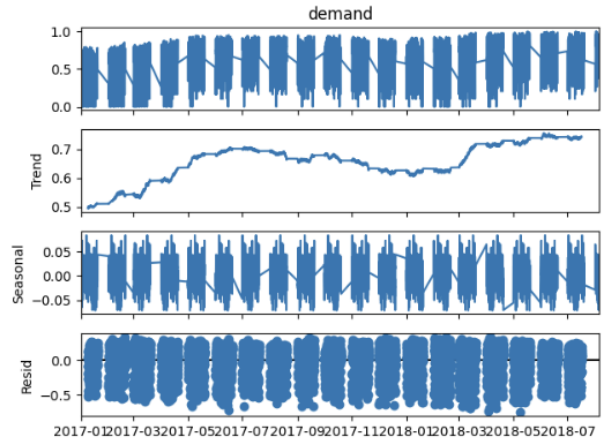

Analyzing Humidity:

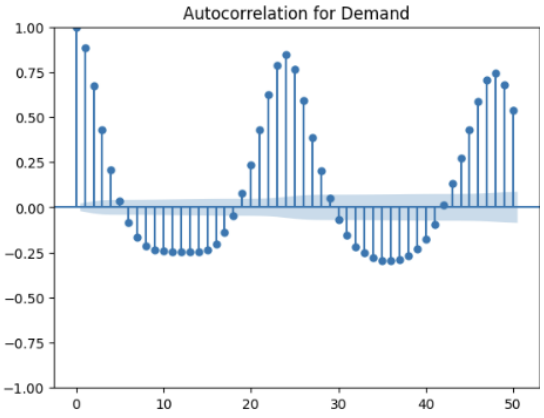



Analyzing Windspeed:



```
Analyzing Demand:
```




```
Summary of Findings:
1. Temperature shows a clear seasonal pattern, with higher values in summer and lower in winter.
2. Humidity does not display a strong seasonal pattern but varies more randomly over the year.
3. Windspeed shows some seasonality, with higher speeds observed in certain months.
4. Demand appears to have a strong seasonal component, peaking in certain seasons and lower in others.
5. Autocorrelation plots reveal significant correlations at specific lags for temperature and demand, indicating seasonality.
6. The decomposition plots for each variable help to visually separate the trend, seasonal, and residual components, providing insights into
the underlying patterns in the data.
```

**4)**

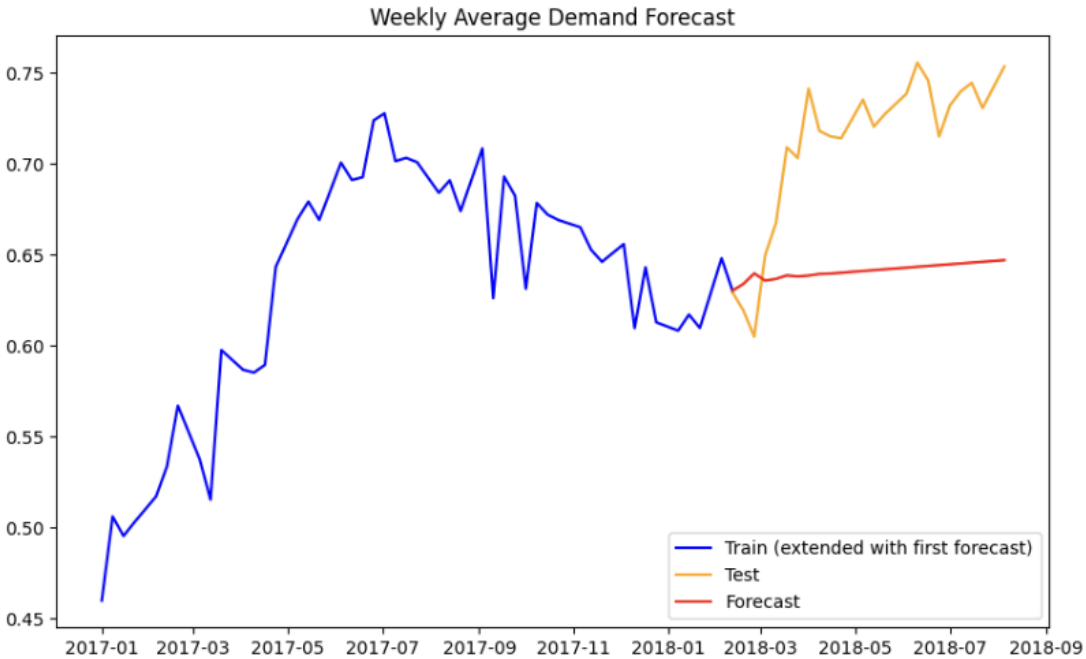
```
import pandas as pd
import numpy as np
from statsmodels.tsa.stattools import adfuller
import matplotlib.pyplot as plt
from pmdarima import auto_arima
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

# Load the preprocessed DataFrame
df = pd.read_csv('CarSharingDataAnalytics.csv')

# Ensure 'timestamp' is a datetime type and set as index
df['timestamp'] = pd.to_datetime(df['timestamp'])
df.set_index('timestamp', inplace=True)
```

```
# Check and handle NaNs in 'demand' column before aggregation
if df['demand'].isnull().any():
    df['demand'] = df['demand'].interpolate()

# Aggregate to weekly average demand
weekly_demand = df['demand'].resample('W').mean()

# After resampling, check again for NaNs and handle them
if weekly_demand.isnull().any():
    weekly_demand = weekly_demand.interpolate()

# Split into training and testing sets (70% train, 30% test)
split_point = int(len(weekly_demand) * 0.7)
train, test = weekly_demand[:split_point], weekly_demand[split_point:]

# Stationarity check using Augmented Dickey-Fuller test
adf_result = adfuller(train)
print(f'ADF Statistic: {adf_result[0]}')
print(f'p-value: {adf_result[1]}')

# Identify ARIMA parameters using auto_arima (this might take some time)
auto_arima_model = auto_arima(train, start_p=0, start_q=0, max_p=5, max_q=5,
seasonal=False, trace=True, error_action='ignore', suppress_warnings=True)

# Display the recommended order
print(f'Recommended ARIMA Order: {auto_arima_model.order}')

# Fit ARIMA model
model = ARIMA(train, order=auto_arima_model.order)
model_fit = model.fit()
```

```
# Forecast
forecasts = model_fit.forecast(steps=len(test))

# Evaluate forecasts
mse = mean_squared_error(test, forecasts)
rmse = sqrt(mse)
print(f'Test RMSE: {rmse}')

# Plot forecasts against actual outcomes with adjustment to connect lines
plt.figure(figsize=(10, 6))
# Extend the training series with the first forecasted value to visually connect the lines
train_extended = np.append(train, forecasts.iloc[0] if isinstance(forecasts, pd.Series) else forecasts[0])
# Plot the extended training data and test data
plt.plot(pd.date_range(start=train.index[0], periods=len(train_extended), freq='W'), train_extended, label='Train (extended with first forecast)', color='blue')
plt.plot(test.index, test, label='Test', color='orange')
plt.plot(test.index, forecasts, label='Forecast', color='red')
plt.title('Weekly Average Demand Forecast')
plt.legend()
plt.show()

# Overwrite the original CSV file with the current state of the DataFrame (even if unchanged)
df.to_csv('CarSharingDataAnalytics.csv', index=True)
```

**Output:**

```
ADF Statistic: -1.9528720170329528
p-value: 0.3076114019162724
Performing stepwise search to minimize aic
 ARIMA(0,2,0)(0,0,0)[0] intercept   : AIC=-189.355, Time=0.04 sec
 ARIMA(1,2,0)(0,0,0)[0] intercept   : AIC=-210.705, Time=0.04 sec
 ARIMA(0,2,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.06 sec
 ARIMA(0,2,0)(0,0,0)[0]             : AIC=-191.348, Time=0.01 sec
 ARIMA(2,2,0)(0,0,0)[0] intercept   : AIC=-232.400, Time=0.03 sec
 ARIMA(3,2,0)(0,0,0)[0] intercept   : AIC=-236.261, Time=0.09 sec
 ARIMA(4,2,0)(0,0,0)[0] intercept   : AIC=-241.860, Time=0.07 sec
 ARIMA(5,2,0)(0,0,0)[0] intercept   : AIC=-246.505, Time=0.13 sec
 ARIMA(5,2,1)(0,0,0)[0] intercept   : AIC=-247.295, Time=0.32 sec
 ARIMA(4,2,1)(0,0,0)[0] intercept   : AIC=-248.644, Time=0.18 sec
 ARIMA(3,2,1)(0,0,0)[0] intercept   : AIC=-249.485, Time=0.17 sec
 ARIMA(2,2,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.14 sec
 ARIMA(3,2,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.20 sec
 ARIMA(2,2,2)(0,0,0)[0] intercept   : AIC=-247.013, Time=0.19 sec
 ARIMA(4,2,2)(0,0,0)[0] intercept   : AIC=-245.858, Time=0.24 sec
 ARIMA(3,2,1)(0,0,0)[0]             : AIC=-250.403, Time=0.13 sec
 ARIMA(2,2,1)(0,0,0)[0]             : AIC=-252.184, Time=0.07 sec
 ARIMA(1,2,1)(0,0,0)[0]             : AIC=-246.107, Time=0.08 sec
 ARIMA(2,2,0)(0,0,0)[0]             : AIC=-234.399, Time=0.02 sec
 ARIMA(2,2,2)(0,0,0)[0]             : AIC=-251.279, Time=0.15 sec
 ARIMA(1,2,0)(0,0,0)[0]             : AIC=-212.703, Time=0.02 sec
 ARIMA(1,2,2)(0,0,0)[0]             : AIC=-251.405, Time=0.14 sec
 ARIMA(3,2,0)(0,0,0)[0]             : AIC=-238.260, Time=0.08 sec
 ARIMA(3,2,2)(0,0,0)[0]             : AIC=-248.210, Time=0.14 sec

Best model:  ARIMA(2,2,1)(0,0,0)[0]
Total fit time: 2.765 seconds
Recommended ARIMA Order: (2, 2, 1)
Test RMSE: 0.0805041154106365
```



**5)**

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt

# Load the preprocessed DataFrame
df = pd.read_csv('CarSharingDataAnalytics.csv')

# Convert 'timestamp' to datetime, if it exists
if 'timestamp' in df.columns:
    df['timestamp'] = pd.to_datetime(df['timestamp'])
    df['year'] = df['timestamp'].dt.year
    df['month'] = df['timestamp'].dt.month
    df['day'] = df['timestamp'].dt.day
    df['weekday'] = df['timestamp'].dt.weekday
    df['hour'] = df['timestamp'].dt.hour
    df.drop('timestamp', axis=1, inplace=True)

# Assuming the dataset is already preprocessed and one-hot encoded as needed

# Split data into features (X) and target variable (y)
X = df.drop('demand', axis=1)
y = df['demand']
```

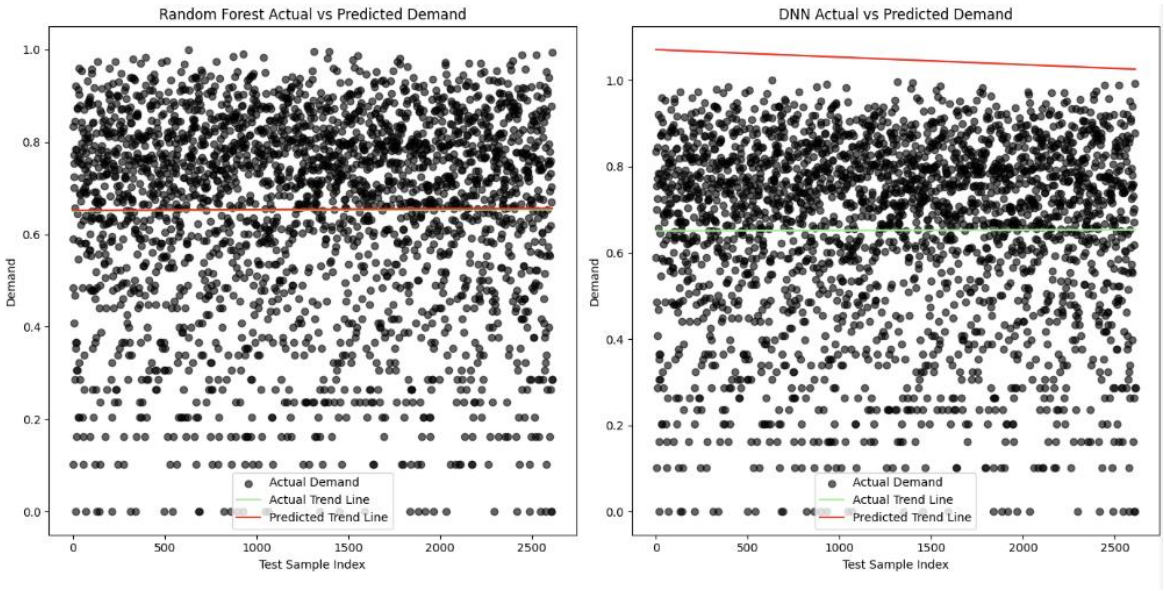
```
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)


# Casting the feature sets to float32 for TensorFlow compatibility
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')


### Random Forest Regressor ###
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
rf_predictions = rf_model.predict(X_test)
rf_mse = mean_squared_error(y_test, rf_predictions)
print(f'Random Forest MSE: {rf_mse:.4f}')


### Deep Neural Network ###
dnn_model = Sequential([
    Input(shape=(X_train.shape[1],)),
    Dense(10, activation='relu'),
    Dense(10, activation='relu'),
    Dense(1)
])
dnn_model.compile(optimizer='adam', loss='mean_squared_error')
early_stopping = EarlyStopping(patience=3)
dnn_model.fit(X_train, y_train, epochs=50, validation_split=0.2,
callbacks=[early_stopping], verbose=0)
dnn_predictions = dnn_model.predict(X_test).flatten()
dnn_mse = mean_squared_error(y_test, dnn_predictions)
print(f'DNN MSE: {dnn_mse:.4f}')
```

```
# Comparing MSE
if rf_mse < dnn_mse:
    print("Random Forest performs better with a lower MSE.")
else:
    print("DNN performs better with a lower MSE.")


# Formatting Actual vs Predicted with pandas DataFrame for a neat presentation
actual_vs_predicted_rf = pd.DataFrame({'Actual': y_test.values, 'RF_Predicted':
rf_predictions})
actual_vs_predicted_dnn = pd.DataFrame({'Actual': y_test.values, 'DNN_Predicted':
dnn_predictions})


print("\nSample Actual vs. Predicted Values (Random Forest):")
print(actual_vs_predicted_rf.head())
print("\nSample Actual vs. Predicted Values (DNN):")
print(actual_vs_predicted_dnn.head())


# Function to plot with trend lines for actual and predicted values in subplots
def plot_with_trendline(ax, actual, predictions, title):
    ax.scatter(np.arange(len(actual)), actual, label='Actual Demand', alpha=0.6,
color='black')
    # Trend line for actual values
    z_actual = np.polyfit(np.arange(len(actual)), actual, 1)
    p_actual = np.poly1d(z_actual)
    ax.plot(np.arange(len(actual)), p_actual(np.arange(len(actual))),
color='lightgreen', label='Actual Trend Line')
    # Trend line for predicted values
    z_predicted = np.polyfit(np.arange(len(predictions)), predictions, 1)
    p_predicted = np.poly1d(z_predicted)
    ax.plot(np.arange(len(predictions)), p_predicted(np.arange(len(predictions))),
color='red', label='Predicted Trend Line')
    ax.set_title(title)
```

```
    ax.set_xlabel('Test Sample Index')
    ax.set_ylabel('Demand')
    ax.legend()

# Plotting for Actual vs Predicted Demand with Trend Lines for both models
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(14, 7))
plot_with_trendline(axs[0], y_test.values, rf_predictions, 'Random Forest Actual vs Predicted Demand')
plot_with_trendline(axs[1], y_test.values, dnn_predictions, 'DNN Actual vs Predicted Demand')
plt.tight_layout()
plt.show()
```

**Output:**

```
Random Forest MSE: 0.0024
82/82 ━━━━━━━━━━━━━━━━━━━━ 0s 772us/step
DNN MSE: 1.0046
Random Forest performs better with a lower MSE.

Sample Actual vs. Predicted Values (Random Forest):
     Actual  RF_Predicted
0  0.589978      0.559834
1  0.834746      0.872222
2  0.102048      0.152573
3  0.485228      0.467939
4  0.754373      0.791741

Sample Actual vs. Predicted Values (DNN):
     Actual  DNN_Predicted
0  0.589978       1.990257
1  0.834746       1.153160
2  0.102048      -0.454384
3  0.485228       0.764976
4  0.754373      -0.747933
```




**6)**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the dataset
print("Loading the dataset...")
df = pd.read_csv('CarSharingDataAnalytics.csv')
print("Dataset loaded successfully.")

# Data Preprocessing and Label Creation
print("Preprocessing data and creating labels...")
average_demand = df['demand'].mean()
df['label'] = df['demand'].apply(lambda x: 1 if x > average_demand else 0)
print(f"Labels created. Average demand: {average_demand:.2f}. Label 1 count:
{sum(df['label'] == 1)}, Label 0 count: {sum(df['label'] == 0)}")
```

```
# Exclude the 'demand' column and the newly created 'label' column from the features
X = df.drop(columns=['demand', 'label'])
y = df['label']

# Split the dataset
print("Splitting the dataset into training and testing sets...")
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
print("Dataset split completed.")

# Model Selection and Training
classifiers = {
    'Decision Tree': DecisionTreeClassifier(),
    'Random Forest': RandomForestClassifier(),
    'SVM': SVC()
}
results = {}

for name, classifier in classifiers.items():
    print(f"\nTraining the {name} model...")
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    results[name] = accuracy
    print(f'{name} model trained successfully. Accuracy: {accuracy:.4f}')

# Finding and Printing the Best Performing Model
best_model = max(results, key=results.get)
print(f"\nBest performing model: {best_model} with an accuracy of {results[best_model]:.4f}.")
```

```
# Optionally, update the original CSV file with the current state of the DataFrame
df.to_csv('CarSharingDataAnalytics_Updated.csv', index=False)
```

**Output:**

```
Loading the dataset...
Dataset loaded successfully.
Preprocessing data and creating labels...
Labels created. Average demand: 0.66. Label 1 count: 5367, Label 0 count: 3341
Splitting the dataset into training and testing sets...
Dataset split completed.

Training the Decision Tree model...
Decision Tree model trained successfully. Accuracy: 0.9059

Training the Random Forest model...
Random Forest model trained successfully. Accuracy: 0.9162

Training the SVM model...
SVM model trained successfully. Accuracy: 0.6705

Best performing model: Random Forest with an accuracy of 0.9162.
```

**7)**

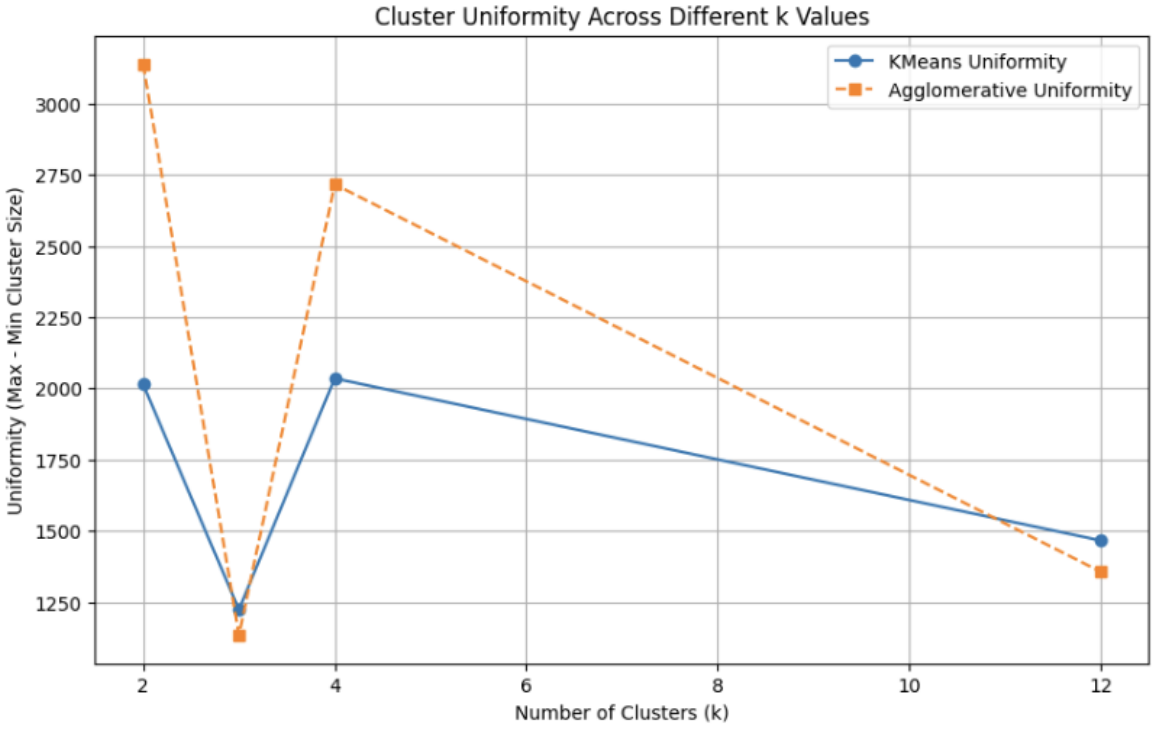
```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans, AgglomerativeClustering
import matplotlib.pyplot as plt

# Load the dataset
print("Loading the dataset...")
df = pd.read_csv('CarSharingDataAnalytics.csv')
print("Dataset loaded successfully.")

# Assuming there's a column named 'temp' for temperature in 2017, adjust accordingly
X = df[['temp']].values
```

```
# Define k values
k_values = [2, 3, 4, 12]

# Initialize dictionaries to hold the cluster labels and uniformity scores for each k
kmeans_labels = {}
agglo_labels = {}
uniformity_kmeans = {}
uniformity_agglo = {}

# Perform clustering with KMeans and Agglomerative Clustering for each k value
for k in k_values:
    # KMeans
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans_labels[k] = kmeans.fit_predict(X)

    # Agglomerative Clustering
    agglo = AgglomerativeClustering(n_clusters=k)
    agglo_labels[k] = agglo.fit_predict(X)

    # Evaluate and print results for KMeans
    unique_kmeans, counts_kmeans = np.unique(kmeans_labels[k],
return_counts=True)
    uniformity_kmeans[k] = max(counts_kmeans) - min(counts_kmeans)

    # Evaluate and print results for Agglomerative Clustering
    unique_agglo, counts_agglo = np.unique(agglo_labels[k], return_counts=True)
    uniformity_agglo[k] = max(counts_agglo) - min(counts_agglo)

# Function to evaluate and print uniformity of clusters
def evaluate_uniformity(labels):
```

```
    unique, counts = np.unique(labels, return_counts=True)
    print(f"Cluster distribution: {dict(zip(unique, counts))}")
    print(f"Most uniform cluster size difference: {max(counts) - min(counts)}")

# Evaluate and print results for each k value for both methods
print("KMeans Clustering Results:")
for k in k_values:
    print(f"\nk = {k}")
    evaluate_uniformity(kmeans_labels[k])

print("\nAgglomerative Clustering Results:")
for k in k_values:
    print(f"\nk = {k}")
    evaluate_uniformity(agglo_labels[k])

# Plotting the uniformity for KMeans and Agglomerative Clustering
plt.figure(figsize=(10, 6))
plt.plot(list(uniformity_kmeans.keys()), list(uniformity_kmeans.values()), marker='o',
linestyle='-', label='KMeans Uniformity')
plt.plot(list(uniformity_agglo.keys()), list(uniformity_agglo.values()), marker='s',
linestyle='--', label='Agglomerative Uniformity')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Uniformity (Max - Min Cluster Size)')
plt.title('Cluster Uniformity Across Different k Values')
plt.legend()
plt.grid(True)
plt.show()
```

**Output:**

```
Loading the dataset...
Dataset loaded successfully.
KMeans Clustering Results:

k = 2
Cluster distribution: {0: 3347, 1: 5361}
Most uniform cluster size difference: 2014

k = 3
Cluster distribution: {0: 2666, 1: 3633, 2: 2409}
Most uniform cluster size difference: 1224

k = 4
Cluster distribution: {0: 2394, 1: 1761, 2: 1259, 3: 3294}
Most uniform cluster size difference: 2035

k = 12
Cluster distribution: {0: 590, 1: 909, 2: 1120, 3: 947, 4: 699, 5: 992, 6: 433, 7: 1558, 8: 334, 9: 92, 10: 603, 11: 431}
Most uniform cluster size difference: 1466

Agglomerative Clustering Results:

k = 2
Cluster distribution: {0: 5923, 1: 2785}
Most uniform cluster size difference: 3138

k = 3
Cluster distribution: {0: 2785, 1: 2394, 2: 3529}
Most uniform cluster size difference: 1135

k = 4
Cluster distribution: {0: 2394, 1: 1972, 2: 3529, 3: 813}
Most uniform cluster size difference: 2716

k = 12
Cluster distribution: {0: 525, 1: 859, 2: 915, 3: 1010, 4: 752, 5: 611, 6: 811, 7: 567, 8: 1558, 9: 202, 10: 408, 11: 490}
Most uniform cluster size difference: 1356
```



**Word Count: 2,915**

**References:**

1.

Fattah, J., Ezzine, L., Aman, Z., El Moussami, H. and Lachhab, A., 2018. Forecasting of demand using ARIMA model. *International Journal of Engineering Business Management*, *10*, p.1847979018808673.