Accessing, Storage, Manipulation and Analysis of Car Sharing data

Hephzibah Akintunde (19013441) | CSC – 40054

Database Management

Part 1

The "CarSharing" table was created in an SQLite database, "CarSharingDB.db," after examining the structure of the CarSharing.csv file, which included columns for id (integer), timestamp, season, holiday, workingday, weather (all string/object), temp, temp_feel, humidity, windspeed, and demand (all float), to understand the data. The database was structured to accommodate these data types (integer, text, real). Following this, data from CarSharing.csv was imported into "CarSharing," and a backup table "CarSharingBackup" was created with the same structure. Initially empty, "CarSharingBackup" was then populated with the "CarSharing" data, ensuring it mirrored the original table's data. Both "CarSharing" and "CarSharingBackup" tables were verified to have 8708 rows, confirming successful data copying.

```
# Create the CarSharing table
create_table_query = '''
CREATE TABLE CarSharing (
    id INTEGER PRIMARY KEY,
    timestamp TEXT,
    season TEXT,
    holiday TEXT,
    workingday TEXT,
    weather TEXT,
    temp REAL,
    temp_feel REAL,
    humidity REAL,
    windspeed REAL,
    demand REAL
)
'''
cursor.execute(create_table_query)

# Import data into the CarSharing table
car_sharing_df.to_sql('CarSharing', conn, if_exists='replace', index=False)

# Create a backup table with the same structure as CarSharing
cursor.execute('CREATE TABLE CarSharingBackup AS SELECT * FROM CarSharing WHERE 1=0')

# Copy all data from CarSharing to CarSharingBackup
cursor.execute('INSERT INTO CarSharingBackup SELECT * FROM CarSharing')

# Commit the changes and close the connection
conn.commit()

# Verify by counting the rows in both tables
row_count_carsharing = cursor.execute('SELECT COUNT(*) FROM CarSharing').fetchone()[0]
row_count_backup = cursor.execute('SELECT COUNT(*) FROM CarSharingBackup').fetchone()[0]

conn.close()

print(row_count_carsharing, row_count_backup)
```

Figure 1. Code for creating 'Carsharing' table and backup table along with verification.

```
8708 8708
```

Figure 2. Output of the code showing the number of rows both the 'Carsharing' table and the backup table.

Part 2

A "temp_category" TEXT column was added to the "CarSharing" table to categorize temperature conditions using SQL queries based on "temp_feel" values: "Cold" for <10°C, "Mild" for 10°C to 25°C, and "Hot" for >25°C. Verification of this addition involved inspecting the first 10 rows for "temp_feel" and "temp_category" values, confirming the correct reflection of temperature categories. This process ensured that the "temp_category" column accurately represents temperature conditions in a textual format, based on the "feels-like" temperature, with the verification confirming its successful implementation.

```
# Step 1: Add the new column "temp_category" to the "CarSharing" table
cursor.execute('ALTER TABLE CarSharing ADD COLUMN temp_category TEXT')

# Step 2: Update the "temp_category" column based on "temp_feel" values
cursor.execute('UPDATE CarSharing SET temp_category = "Cold" WHERE temp_feel < 10')
cursor.execute('UPDATE CarSharing SET temp_category = "Mild" WHERE temp_feel >= 10 AND temp_feel <= 25')
cursor.execute('UPDATE CarSharing SET temp_category = "Hot" WHERE temp_feel > 25')

# Commit the changes
conn.commit()

# Step 3: Verification - Select a few rows to check the "temp_category" values
verification_query = 'SELECT id, temp_feel, temp_category FROM CarSharing LIMIT 10'
verification_results = cursor.execute(verification_query).fetchall()

conn.close()

print(verification_results)
```

Figure 3. Code showing the addition and population of 'temp_category' column along with verification of results.

```
[(1, 14.395, 'Mild'), (2, 13.635, 'Mild'), (3, 13.635, 'Mild'), (4, 14.39
5, 'Mild'), (5, 14.395, 'Mild'), (6, 12.88, 'Mild'), (7, 13.635, 'Mild'),
(8, 12.88, 'Mild'), (9, 14.395, 'Mild'), (10, 17.425, 'Mild')]
```

Figure 4. Output of verification code showing first 10 rows of 'temp_category' data.

Part 3

A "temperature" table with columns "id", "temp", "temp_feel", and "temp_category" was created and populated from the "CarSharing" table. Due to SQLite limitations, a "CarSharing_temp" table excluded "temp" and "temp_feel" columns. After dropping the original "CarSharing", "CarSharing_temp" was renamed to "CarSharing", removing "temp" and "temp_feel". Verification confirmed the "temperature" table's structure and the updated "CarSharing" table, now including "id", "timestamp", "season", "holiday", "workingday", "weather", "humidity", "windspeed", "demand", and "temp_category", confirming the successful column removal and restructuring.

```
# Step 1: Create the "temperature" table
cursor.execute('CREATE TABLE temperature AS SELECT id, temp, temp_feel, temp_category FROM CarSharing')

# Step 2: Remove the "temp" and "temp_feel" columns from the "CarSharing" table
# 2.1 Create a temporary table excluding the "temp" and "temp_feel" columns
cursor.execute('''
CREATE TABLE CarSharing_temp AS
SELECT id, timestamp, season, holiday, workingday, weather, humidity, windspeed, demand, temp_category
FROM CarSharing
''')

# 2.2 Drop the original "CarSharing" table
cursor.execute('DROP TABLE CarSharing')

# 2.3 Rename the temporary table to "CarSharing"
cursor.execute('ALTER TABLE CarSharing_temp RENAME TO CarSharing')

# Commit the changes
conn.commit()

# Step 3: Verification
# 3.1 Verify the "temperature" table structure
temperature_table_structure = cursor.execute('PRAGMA table_info(temperature)').fetchall()

# 3.2 Verify the modified "CarSharing" table structure
carsharing_table_structure = cursor.execute('PRAGMA table_info(CarSharing)').fetchall()

conn.close()

print(temperature_table_structure, carsharing_table_structure)
```

Figure 5. Code showing creation of temperature table and modification of 'CarSharing' table.

```
[(0, 'id', 'INT', 0, None, 0), (1, 'temp', 'REAL', 0, None, 0), (2, 'temp_feel', 'REAL', 0, Non
e, 0), (3, 'temp_category', 'TEXT', 0, None, 0)] [(0, 'id', 'INT', 0, None, 0), (1, 'timestamp'
, 'TEXT', 0, None, 0), (2, 'season', 'TEXT', 0, None, 0), (3, 'holiday', 'TEXT', 0, None, 0), (
4, 'workingday', 'TEXT', 0, None, 0), (5, 'weather', 'TEXT', 0, None, 0), (6, 'humidity', 'REAL
', 0, None, 0), (7, 'windspeed', 'REAL', 0, None, 0), (8, 'demand', 'REAL', 0, None, 0), (9, 't
emp_category', 'TEXT', 0, None, 0)]
```

Figure 6. Output of verification code showing both temperature and 'CarSharing' table information.

Part 4

Distinct weather conditions in the "CarSharing" table were identified as "Clear or partly cloudy", "Mist", "Light snow or rain", and "heavy rain/ice pellets/snow + fog", each assigned a unique integer code: 1, 2, 3, and 4, respectively. A new "weather_code" column was added to store these codes. This column was populated using the established mappings, linking weather conditions to their corresponding codes. Verification involved selecting the first 10 rows to check "weather" and "weather_code" values, confirming accurate population of the "weather_code" column according to the predefined assignments.

```
# Step 1: Identify distinct weather values
distinct_weather = cursor.execute('SELECT DISTINCT weather FROM CarSharing').fetchall()
distinct_weather = [weather[0] for weather in distinct_weather]

# Step 2: Assign a unique number to each weather condition
weather_codes = {weather: code for code, weather in enumerate(distinct_weather, start=1)}

# Step 3: Add the "weather_code" column to the "CarSharing" table
cursor.execute('ALTER TABLE CarSharing ADD COLUMN weather_code INTEGER')

# Step 4: Update the "weather_code" column based on the weather condition
for weather, code in weather_codes.items():
    cursor.execute('UPDATE CarSharing SET weather_code = ? WHERE weather = ?', (code, weather))

# Commit the changes
conn.commit()

# Step 5: Verification - Select a few rows to check the "weather" and "weather_code" values
verification_query = 'SELECT id, weather, weather_code FROM CarSharing LIMIT 10'
verification_results = cursor.execute(verification_query).fetchall()

conn.close()

print(weather_codes, verification_results)
```

Figure 7. Code showing identification of distinct weather conditions as well as addition and population of 'weather_code' column.

```
{'Clear or partly cloudy': 1, 'Mist': 2, 'Light snow or rain': 3, 'heavy rain/ice pellets/snow
+ fog': 4} [(1, 'Clear or partly cloudy', 1), (2, 'Clear or partly cloudy', 1), (3, 'Clear or p
artly cloudy', 1), (4, 'Clear or partly cloudy', 1), (5, 'Clear or partly cloudy', 1), (6, 'Mis
t', 2), (7, 'Clear or partly cloudy', 1), (8, 'Clear or partly cloudy', 1), (9, 'Clear or partl
y cloudy', 1), (10, 'Clear or partly cloudy', 1)]
```

Figure 8. Output of verification code showing first 10 rows of weather and weather code information.

Part 5

A "weather" table with "weather" and "weather_code" columns was created to hold unique weather conditions and codes, eliminating duplicates. Due to SQLite's column dropping limitations, a "CarSharing_temp" table excluded the "weather" column. After dropping the original "CarSharing", "CarSharing_temp" was renamed to "CarSharing", removing the "weather" column. The "weather" table's structure, containing "weather" and "weather_code", was confirmed. The updated "CarSharing" now includes "id", "timestamp", "season", "holiday", "workingday", "humidity", "windspeed", "demand", "temp_category", and "weather_code", verifying the successful column removal and table modification.

```
# Step 1: Create the "weather" table with unique pairs of "weather" conditions and "weather_code"
cursor.execute('CREATE TABLE weather AS SELECT DISTINCT weather, weather_code FROM CarSharing')

# Step 3: Remove the "weather" column from the "CarSharing" table using the workaround
# 3.1 Create a temporary table without the "weather" column
cursor.execute('''
CREATE TABLE CarSharing_temp AS
SELECT id, timestamp, season, holiday, workingday, humidity, windspeed, demand, temp_category, weather_code
FROM CarSharing
''')

# 3.2 Drop the original "CarSharing" table
cursor.execute('DROP TABLE CarSharing')

# 3.3 Rename the temporary table to "CarSharing"
cursor.execute('ALTER TABLE CarSharing_temp RENAME TO CarSharing')

# Commit the changes
conn.commit()

# Step 4: Verification
# 4.1 Verify the "weather" table structure
weather_table_structure = cursor.execute('PRAGMA table_info(weather)').fetchall()

# 4.2 Verify the modified "CarSharing" table structure
carsharing_table_structure = cursor.execute('PRAGMA table_info(CarSharing)').fetchall()

conn.close()

print(weather_table_structure, carsharing_table_structure)
```

Figure 9. Code showing creation of 'weather' table and modification of 'CarSharing' table as well as verification.

```
[(0, 'weather', 'TEXT', 0, None, 0), (1, 'weather_code', 'INT', 0, None, 0)] [(0, 'id', 'INT',
0, None, 0), (1, 'timestamp', 'TEXT', 0, None, 0), (2, 'season', 'TEXT', 0, None, 0), (3, 'holi
day', 'TEXT', 0, None, 0), (4, 'workingday', 'TEXT', 0, None, 0), (5, 'humidity', 'REAL', 0, No
ne, 0), (6, 'windspeed', 'REAL', 0, None, 0), (7, 'demand', 'REAL', 0, None, 0), (8, 'temp_cate
gory', 'TEXT', 0, None, 0), (9, 'weather_code', 'INT', 0, None, 0)]
```

Figure 10. Output of verification showing the structure of both the 'weather' and modified 'CarSharing' table.

Part 6

The SQLite function strftime() was used to extract hours as integers and map weekdays and months by name within an SQL query during table creation for efficient data transformation. A "time" table was created, featuring columns for the original timestamp, extracted hour, weekday name, and month name from each "CarSharing" table record, populated via a combined SQL statement for table creation, data extraction, and mapping. Verification involved selecting the "time" table's first 10 rows, confirming accurate columns for timestamp, hour, weekday, and month names based on the original timestamps, e.g., a timestamp "2017-01-01 00:00:00" accurately showing 0 hour, "Sunday", and "January".

```
# Step 1 & 2: Create the "time" table with timestamp, hour, weekday name, and month name
# SQLite treats Monday as 1, hence the adjustment in weekday calculation to match common expectations
cursor.execute('''
CREATE TABLE time AS
SELECT
    timestamp,
    CAST(strftime('%H', timestamp) AS INTEGER) AS hour,
    CASE CAST(strftime('%w', timestamp) AS INTEGER)
        WHEN 0 THEN 'Sunday'
        WHEN 1 THEN 'Monday'
        WHEN 2 THEN 'Tuesday'
        WHEN 3 THEN 'Wednesday'
        WHEN 4 THEN 'Thursday'
        WHEN 5 THEN 'Friday'
        WHEN 6 THEN 'Saturday'
    END AS weekday,
    CASE CAST(strftime('%m', timestamp) AS INTEGER)
        WHEN 1 THEN 'January'
        WHEN 2 THEN 'February'
        WHEN 3 THEN 'March'
        WHEN 4 THEN 'April'
        WHEN 5 THEN 'May'
        WHEN 6 THEN 'June'
        WHEN 7 THEN 'July'
        WHEN 8 THEN 'August'
        WHEN 9 THEN 'September'
        WHEN 10 THEN 'October'
        WHEN 11 THEN 'November'
        WHEN 12 THEN 'December'
    END AS month
FROM CarSharing
''')

# Commit the changes
conn.commit()

# Step 4: Verification - Select a few rows to check the "time" table structure and data
verification_query = 'SELECT * FROM time LIMIT 10'
verification_results = cursor.execute(verification_query).fetchall()

conn.close()

print(verification_results)
```

Figure 11. Code showing creation of 'time' table along with verification.

```
[('2017-01-01 00:00:00', 0, 'Sunday', 'January'), ('2017-01-01 01:00:00', 1, 'Su
nday', 'January'), ('2017-01-01 02:00:00', 2, 'Sunday', 'January'), ('2017-01-01
 03:00:00', 3, 'Sunday', 'January'), ('2017-01-01 04:00:00', 4, 'Sunday', 'Janua
ry'), ('2017-01-01 05:00:00', 5, 'Sunday', 'January'), ('2017-01-01 06:00:00', 6
, 'Sunday', 'January'), ('2017-01-01 07:00:00', 7, 'Sunday', 'January'), ('2017-
01-01 08:00:00', 8, 'Sunday', 'January'), ('2017-01-01 09:00:00', 9, 'Sunday', '
January')]
```

Figure 12. Output of verification showing first 10 rows of information from the 'time' table.

Part 7

Part (a)

Querying the "CarSharing" table revealed the highest demand rate in 2017 was on June 15 at 17:00, with a demand rate of approximately 6.46, inferred to be on a transformed scale (e.g., log scale).

```
# Part (a): Highest demand rate date and time in 2017
highest_demand_query = '''
SELECT timestamp, MAX(demand) as max_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
'''
highest_demand_result = cursor.execute(highest_demand_query).fetchone()

print(highest_demand_result)
```

Figure 13. Code showing the querying of the 'CarSharing' table to obtain highest demand rate.

```
('2017-06-15 17:00:00', 6.45833828334479)
```

Figure 14. Output of high demand rate query code showing the date and time of the highest demand rate.

Part (b)

After joining and aggregating tables to include weekday, month, season, and demand, analysis for 2017, the highest and lowest average demand rates were identified. The peak was observed on Sundays in July (Fall) with an average demand of approximately 5.00. Conversely, the lowest demand occurred on Mondays in January (Spring), with an average of approximately 3.05.

```
# Part (b): Weekday, Month, and Season with Highest and Lowest Average Demand Rates in 2017
average_demand_query = '''
SELECT t.weekday, t.month, c.season, AVG(c.demand) as avg_demand
FROM CarSharing c
JOIN time t ON c.timestamp = t.timestamp
WHERE strftime('%Y', c.timestamp) = '2017'
GROUP BY t.weekday, t.month, c.season
ORDER BY avg_demand DESC
'''

average_demand_results = cursor.execute(average_demand_query).fetchall()

# Extracting the rows with the highest and lowest average demand
highest_average_demand = average_demand_results[0]
lowest_average_demand = average_demand_results[-1]

print(highest_average_demand, lowest_average_demand)
```

Figure 15. Code showing creation of table containing name of the weekday, month, and season in which the highest and lowest average demand rates occurred throughout 2017 average demand.

```
('Sunday', 'July', 'fall', 4.997135078747038) ('Monday', 'January', 'spring', 3.
0507857781010803)
```

Figure 16. Output showing the highest and lowest average demand rates from table.

Part (c)

After identifying Sunday as the day with the highest average demand from part (b), a table was generated to display the average demand rate by hour for Sundays throughout 2017, organised

in descending order by average demand. This analysis revealed peak demand in the afternoon, with the highest average at 15:00 hours around 5.54. Demand noticeably decreases moving from afternoon to evening and drops to its lowest in the early morning, with 05:00 hours marking the minimum average demand at approximately 1.65.

```
# Part (c): Average Demand Rate at Different Hours for Sunday throughout 2017
average_demand_by_hour_query = '''
SELECT t.hour, AVG(c.demand) as avg_demand
FROM CarSharing c
JOIN time t ON c.timestamp = t.timestamp
WHERE strftime('%Y', c.timestamp) = '2017' AND t.weekday = 'Sunday'
GROUP BY t.hour
ORDER BY avg_demand DESC
'''

average_demand_by_hour_results = cursor.execute(average_demand_by_hour_query).fetchall()

print(average_demand_by_hour_results)
```

Figure 17. Code showing creation of average demand rate by hour table for Sundays throughout 2017.

```
[(15, 5.537925196766501), (14, 5.513702656176965), (16, 5.496274498482708), (13,
 5.478758230367251), (12, 5.457459126923962), (17, 5.367634349646671), (11, 5.29
0915067582866), (18, 5.241211627173081), (10, 5.074545768738012), (19, 5.0418024
30032878), (20, 4.790666498034493), (9, 4.683176378243488), (21, 4.6042216294700
635), (22, 4.47761997279496), (23, 4.3467540044988375), (8, 4.204916044052769),
(0, 4.1349741021373445), (1, 3.8698532861655472), (2, 3.6112321135368823), (7, 3
.2940243626024213), (3, 2.7702626411657163), (6, 2.453218590596452), (4, 1.65922
73354933094), (5, 1.649176214758796)]
```

Figure 18. Output of printing the Sunday average demand rate table.

Part (d)

In 2017, the "Mild" temperature category was most frequent with 2660 instances, and "Clear or partly cloudy" was the common weather condition, occurring 3583 times. Wind speed and humidity data from January to December reveal seasonal variations, such as February's peak wind speed of 52 units and September's highest average humidity at 74.84%. Analysis of average demand rates by weather condition shows a trend: "Hot" weather conditions led to an average demand rate of 4.80, "Mild" conditions had an average of 4.02, and "Cold" conditions saw the lowest average demand rate at 3.19, indicating higher demand in warmer weather.

```
# Part (d) Weather overview in 2017: Predominant temperature category and prevalent weather condition
temperature_category_query = '''
SELECT temp_category, COUNT(temp_category) as count
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
GROUP BY temp_category
ORDER BY count DESC
LIMIT 1
'''

most_prevalent_weather_condition_query = '''
SELECT weather, COUNT(weather) as count
FROM weather
JOIN CarSharing ON weather.weather_code = CarSharing.weather_code
WHERE strftime('%Y', CarSharing.timestamp) = '2017'
GROUP BY weather
ORDER BY count DESC
LIMIT 1
'''

# Average, highest, and lowest wind speed and humidity for each month in 2017
wind_speed_humidity_query = '''
SELECT
    strftime('%m', timestamp) as month,
    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
ORDER BY month
'''

# Average demand rate for each weather condition in 2017
average_demand_by_weather_condition_query = '''
SELECT temp_category, AVG(demand) as avg_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
GROUP BY temp_category
ORDER BY avg_demand DESC
'''

# Execute queries
predominant_temperature_category = cursor.execute(temperature_category_query).fetchone()
most_prevalent_weather_condition = cursor.execute(most_prevalent_weather_condition_query).fetchone()
wind_speed_humidity_stats = cursor.execute(wind_speed_humidity_query).fetchall()
average_demand_by_weather_condition = cursor.execute(average_demand_by_weather_condition_query).fetchall()

print(predominant_temperature_category)
print(most_prevalent_weather_condition)
print(wind_speed_humidity_stats)
print(average_demand_by_weather_condition)
```

Figure 18. Code for 2017 weather overview and production of average demand rate for each weather condition table.

```
('Mild', 2660)
```

Figure 19. Output for predominant temperature category.

```
('Clear or partly cloudy', 3583)
```

Figure 20. Output for most prevalent weather condition.

```
[('01', 13.748052358490567, 39.0007, 0.0, 56.30769230769231, 100.0, 28.0), ('02'
, 15.577716628175509, 51.9987, 0.0, 53.58071748878924, 100.0, 8.0), ('03', 15.97
4884101382482, 40.9973, 0.0, 55.997752808988764, 100.0, 0.0), ('04', 15.85227511
2107584, 40.9973, 0.0, 66.2488986784141, 100.0, 22.0), ('05', 12.427390827740476
, 40.9973, 0.0, 71.37142857142857, 100.0, 24.0), ('06', 11.827618161434977, 35.0
008, 0.0, 58.370860927152314, 100.0, 20.0), ('07', 12.015845657015571, 56.9969,
0.0, 60.29203539823009, 94.0, 17.0), ('08', 12.411122347629782, 43.0006, 0.0, 62
.17362637362638, 94.0, 25.0), ('09', 11.564080089485438, 40.9973, 0.0, 74.840354
76718404, 100.0, 42.0), ('10', 10.892052370203167, 36.9974, 0.0, 71.571428571428
57, 100.0, 29.0), ('11', 12.142271171171178, 36.9974, 0.0, 64.16923076923077, 10
0.0, 27.0), ('12', 10.8364595505618, 43.0006, 0.0, 65.18061674008811, 100.0, 26.
0)]
```

Figure 21. Output for wind speed and humidity stats.

```
[('Hot', 4.79817253046063), (None, 4.044827061441365), ('Mild', 4.02101542912621
6), ('Cold', 3.1902527494822346)]
```

Figure 22. Output for average demand by weather condition.

Part (e)

The month with the highest average demand rate for 2017 was determined by querying the CarSharing table to calculate and group the average demand rate for each month, identifying July as the peak month. For July, further queries provided wind speed and humidity statistics: the average wind speed was 12.02, with a maximum of 56.99 and a minimum of 0.0; average humidity was 60.29%, with a maximum of 94.0% and a minimum of 17.0%. The analysis of average demand rates by temperature category for July showed a notable pattern: a significant average demand rate of 4.79 was recorded in the "Hot" category, with the absence of "Mild" and "Cold" categories, highlighting July's predominantly hot conditions. This month-specific analysis reveals detailed insights into the interplay of temperature, wind speed, and humidity with demand rates, offering a focused comparison to the broader annual data and illustrating the impact of seasonal weather variations on demand within the month identified with the highest average demand.

```
# Step 1: Identify the month with the highest average demand rate in 2017
highest_avg_demand_month_query = '''
SELECT strftime('%m', timestamp) AS month, AVG(demand) AS avg_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
GROUP BY month
ORDER BY avg_demand DESC
LIMIT 1
'''
highest_avg_demand_month = cursor.execute(highest_avg_demand_month_query).fetchone()[0]

# Step 2: Retrieve wind speed and humidity statistics for the identified month
wind_speed_humidity_for_month_query = '''
SELECT
    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' AND strftime('%m', timestamp) = ?
'''
wind_speed_humidity_for_month_stats = cursor.execute(wind_speed_humidity_for_month_query, (highest_avg_demand_month,)).fetchall()

# Step 3: Compare average demand rates by temperature category for the identified month
average_demand_by_temp_category_for_month_query = '''
SELECT temp_category, AVG(demand) AS avg_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017' AND strftime('%m', timestamp) = ?
GROUP BY temp_category
ORDER BY avg_demand DESC
'''
average_demand_by_temp_category_for_month = cursor.execute(average_demand_by_temp_category_for_month_query, (highest_avg_demand_month,)).fetchall()

# Close the connection
conn.close()

print(highest_avg_demand_month, wind_speed_humidity_for_month_stats, average_demand_by_temp_category_for_month)
```

Figure 23. Code showing database being queried for required information.

```
07 [(12.015845657015571, 56.9969, 0.0, 60.29203539823009, 94.0, 17.0)] [(None,
4.808050811008408), ('Hot', 4.787336899703814)]
```

Figure 24. Output showing further statistics for identification of July and highest demand rate month for 2017 along with additional statistics from part (d).

Data Analytics

Part 1

Upon analysis, the dataset, containing 8,708 rows and 11 columns, was found to be free of duplicate rows, maintaining its size before and after duplicate removal attempts. However, null values were discovered in several key columns: 'temp' (1,202 nulls), 'temp_feel' (102 nulls), 'humidity' (39 nulls), and 'windspeed' (200 nulls), with other columns being unaffected. The strategy for handling these null values required a nuanced approach, tailored to the nature of each variable and the overall data structure.

```
timestamp        0
season           0
holiday          0
workingday       0
weather          0
temp          1202
temp_feel      102
humidity        39
windspeed      200
demand           0
dtype: int64
```

Figure 24. Output showing number of null values in each data column.

For 'temp' and 'temp_feel', given their numerical nature and the assumption of a normal distribution, the initial consideration was between using the mean or median for imputation. The decision would hinge on the distribution's skewness, but for time-series data, interpolation emerged as a potentially superior method, aiming to preserve the temporal flow and relationships between data points.

Similarly, 'humidity' and 'windspeed' presented numerical data with missing values. Despite the relatively small number of nulls, especially in 'humidity', the choice of imputation method (mean, median, or interpolation) was crucial for ensuring data integrity without disrupting inherent patterns.

Before implementing any null value filling, a preliminary assessment of the distributions for these columns was conducted. Histograms indicated 'temp' and 'temp_feel' had normal distributions with slight skews, suggesting mean or median imputation could be appropriate.

However, the time-series nature of the dataset tilted the preference towards interpolation, aligning with the goal to maintain temporal relationships.

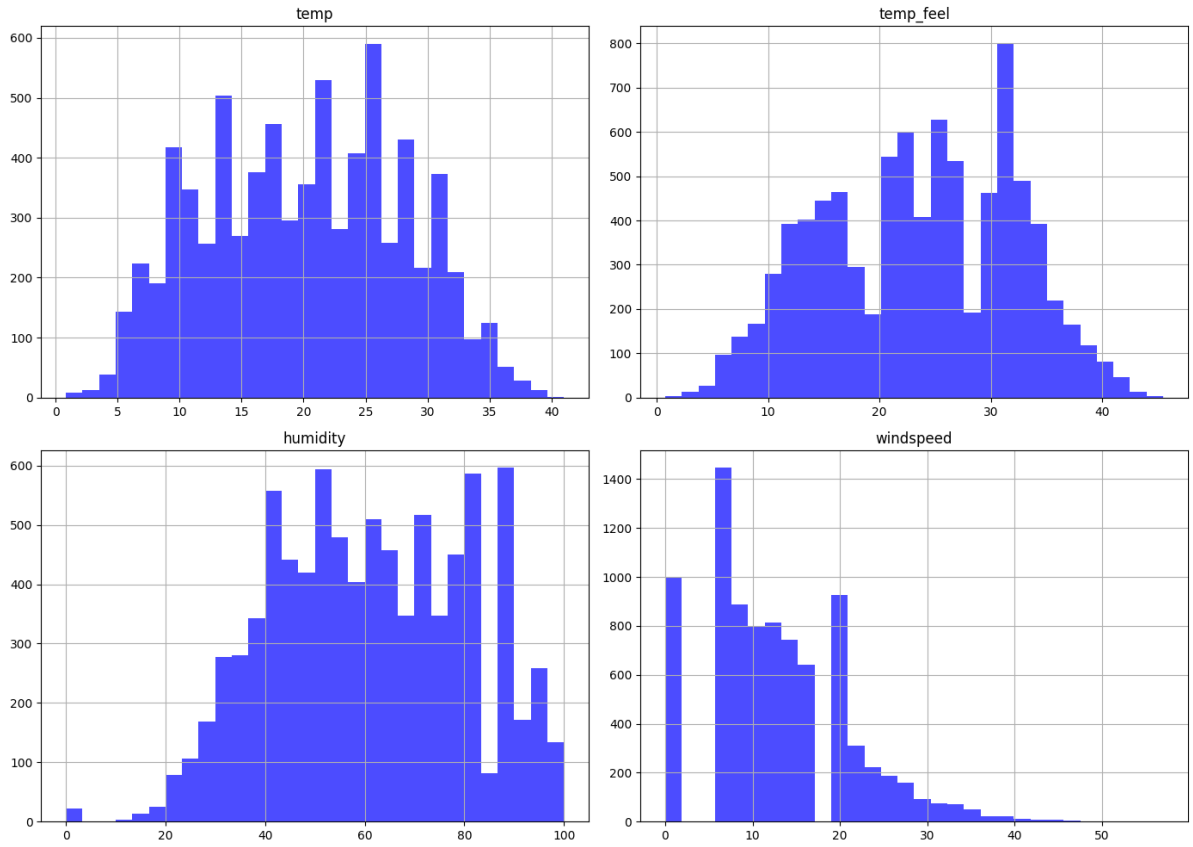

Figure 25. histograms providing insights into the distributions of the columns with null values.

'Humidity' showed a normal distribution with a slight left skew, and 'windspeed' exhibited a right skew with a concentration of lower-end values. In instances of skewed distributions, median imputation might usually be favoured over mean, but again, the sequential aspect of time-series data suggested interpolation as the more fitting choice, aiming to preserve the sequence of data points.

Ultimately, considering the dataset's time-series characteristics, linear interpolation was selected for all affected columns. This method filled null values based on neighbouring points, apt for time-series data where values are interrelated. Post-interpolation, a final verification confirmed the successful addressing of null values across the dataset, ensuring a complete dataset ready for further analysis or modelling efforts.

```
id            0
timestamp     0
season        0
holiday       0
workingday    0
weather       0
temp          0
temp_feel     0
humidity      0
windspeed     0
demand        0
dtype: int64
```

Figure 26. Output showing number of null values in each data column after pre-processing.

Part 2

To determine the relationship between each variable (excluding timestamp) and demand rate, hypothesis tests were conducted. Pearson's correlation coefficient analysed numerical variables' linear correlations with demand, rating them between -1 and 1 to indicate strong positive or negative correlations, or near 0 for no correlation. Numerical variables' hypotheses posited no linear correlation (H0) versus a linear correlation (H1). For categorical variables, ANOVA tests checked for significant demand rate differences across categories, with H0 stating no differences and H1 suggesting at least one category differs significantly.

```
from scipy.stats import pearsonr, f_oneway

# Identifying numerical and categorical columns
numerical_columns = data_interpolated.select_dtypes(include=['float64', 'int64']).columns.drop(['id', 'demand'])
categorical_columns = data_interpolated.select_dtypes(include=['object']).columns.drop(['timestamp'])

# Initializing dictionaries to store test results
pearson_results = {}
anova_results = {}

# Pearson Correlation for Numerical Columns
for col in numerical_columns:
    corr, p_value = pearsonr(data_interpolated[col], data_interpolated['demand'])
    pearson_results[col] = (corr, p_value)

# ANOVA for Categorical Columns
for col in categorical_columns:
    groups = [data_interpolated['demand'][data_interpolated[col] == category] for category in data_interpolated[col].unique()]
    f_stat, p_value = f_oneway(*groups)
    anova_results[col] = (f_stat, p_value)

print(pearson_results, anova_results)
```

Figure 27. Code showing the hypothesis tests carried out using python.

Numerical Columns:

- 'temp': Pearson correlation of 0.40 and p-value of 0.0 indicated a moderate positive relationship, statistically significant.
- 'temp_feel': Pearson correlation of 0.39 with a nearly 0 p-value, showing a moderate positive relationship, significant.
- 'humidity': Pearson correlation of -0.33 and near 0 p-value, showing a moderate negative relationship, significant.

- ‘windspeed’: Pearson correlation of 0.12 and p-value of 4.84e-30, indicating a weak positive relationship, significant.

Categorical Columns:

- ‘season’: ANOVA yielded an F-statistic of 150.06 and p-value of 8.02e−95, demonstrating significant demand variation across seasons.
- ‘holiday’: ANOVA showed an F-statistic of 0.011 and p-value of 0.916, indicating no significant demand difference between holidays and non-holidays.
- ‘workingday’: ANOVA resulted in an F-statistic of 2.87 and p-value of 0.090, suggesting no significant difference in demand, with the p-value marginally close to the alpha level of 0.05.
- ‘weather’: ANOVA showed an F-statistic of 48.59 and p-value of 3.93e−31, indicating significant demand variation across weather conditions.

```
{'temp': (0.39574903874477346, 0.0), 'temp_feel': (0.3929134773035908, 2.1334e-3
19), 'humidity': (-0.33191324669600714, 5.646688203158171e-223), 'windspeed': (0
.12159185707424182, 4.84359823685662 96e-30)} {'season': (150.0648218917323, 8.02
4921568562112e-95), 'holiday': (0.011054437013371764, 0.9162670761960895), 'work
ingday': (2.868367223957404, 0.09037224773166397), 'weather': (48.58618523652949
5, 3.9279297308870713e-31)}
```

Figure 28. Output showing results of the hypothesis tests.

These findings, underscored by p-values, reveal significant relationships between certain variables and demand rate, highlighting the impact of temperature and weather conditions on demand while showing no substantial effect from holiday status or working days.

Part 3

Analysing 2017 data for seasonal or cyclic patterns in temperature, humidity, windspeed, and demand involved first ensuring the timestamp was formatted for time series analysis, setting it as the DataFrame index. This facilitated data slicing by time periods and leveraged pandas' time series capabilities. The dataset was filtered to 2017, and data was resampled monthly to discern trends, with line plots visualising these metrics over time.

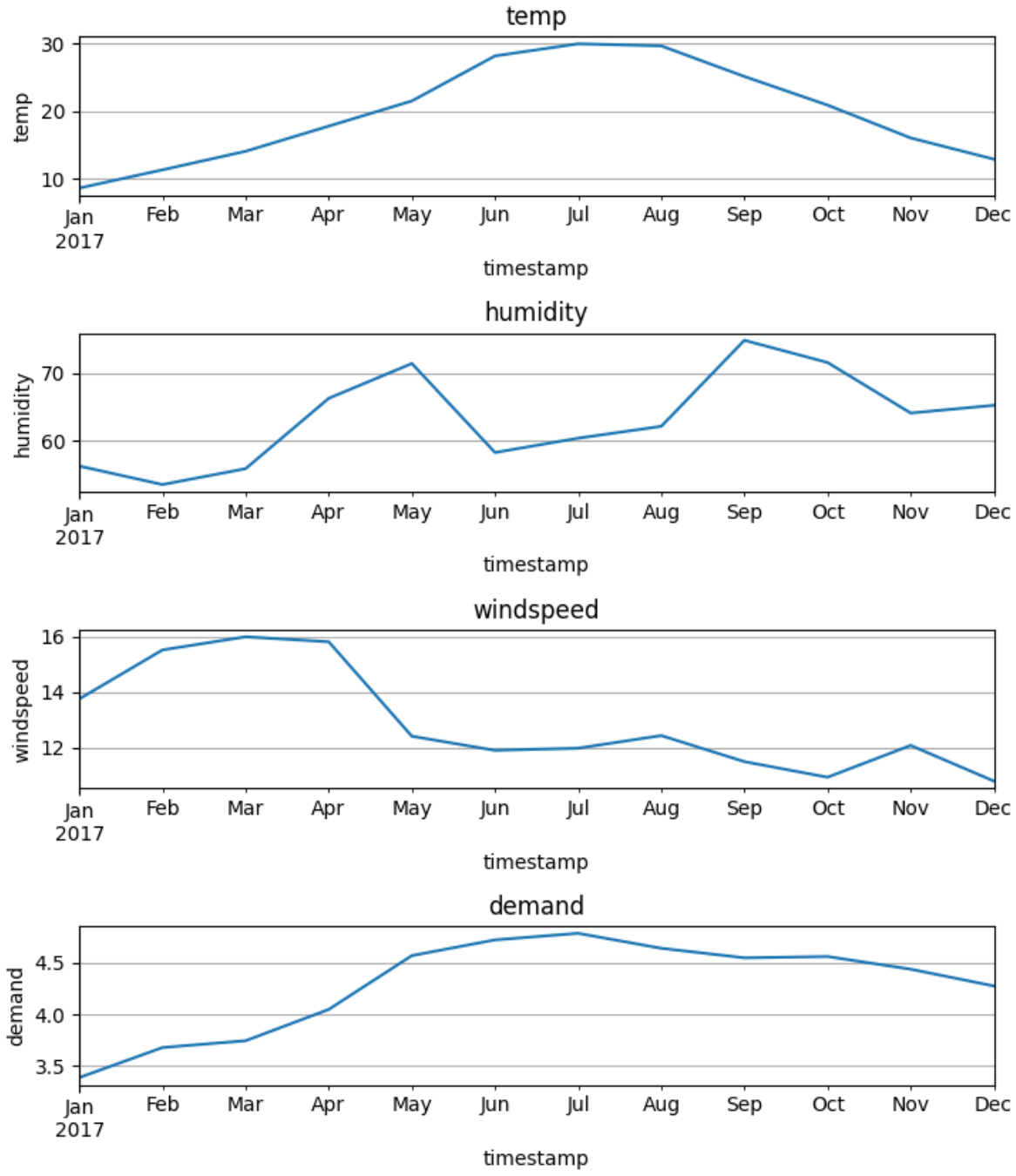

Figure 29. Time series line plots for temp, humidity, windspeed and demand over time.

The analysis unearthed clear seasonal trends:

- Temperature: Exhibited a pronounced seasonal pattern, with peaks in summer months (June, July, August) and lows in winter (January, February, December), mirroring typical seasonal temperature shifts.
- Humidity: Showed seasonal variation, albeit less marked than temperature. Mid-year saw slightly higher humidity, with minor fluctuations throughout, indicating a less clear cyclic pattern.

- Windspeed: Displayed variability without a clear seasonal trend, characterized by irregular peaks and troughs, suggesting no straightforward seasonal influence.
- Demand: Mirrored the temperature pattern, with higher demand in warmer months and reduced demand in cooler times, highlighting a seasonal pattern in car sharing usage influenced by temperature.

These findings spotlight seasonal patterns in temperature and demand, with temperature directly affecting car sharing demand. While humidity presented subtle seasonal shifts, windspeed did not follow a predictable seasonal or cyclic pattern.

Part 4

To forecast the weekly average demand rate, the dataset was first transformed into weekly averages, resulting in a series reflecting the weekly demand. This series was divided into training (71 weeks) and testing sets (31 weeks), adhering to a 70-30 split for model training and evaluation. During preparation, a NaN value in the weekly averages—likely from missing data—was addressed by imputing with the mean weekly demand.

```
timestamp
2017-01-01    3.123272
2017-01-08    3.437559
2017-01-15    3.364685
2017-01-22    3.415826
2017-01-29         NaN
Freq: W-SUN, Name: demand, dtype: float64 (58,) (26,)
```

Figure 30. Output showing first few weekly average demand rate entries to verify transformation. Nan value present.

An ARIMA model with parameters (p=1, d=1, q=2) was selected for its simplicity and adequacy for the data's characteristics, signifying one auto-regressive term, one differencing operation for stationarity, and one moving average term. This choice was grounded in standard practice and observed data traits, though an automated parameter selection via Auto ARIMA could optimise this process.

```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

# Step 1: Prepare Dataset - Convert demand data into weekly averages
weekly_demand = data_interpolated['demand'].resample('W').mean()

# Step 2: Split Dataset into training and testing sets
split_point = int(len(weekly_demand) * 0.7)
train, test = weekly_demand[0:split_point], weekly_demand[split_point:]

# Display the first few entries of the weekly average demand to verify the transformation
print(weekly_demand.head(), train.shape, test.shape)

# Handling NaN value in weekly_demand by filling it with the mean of the series
weekly_demand_filled = weekly_demand.fillna(weekly_demand.mean())

# Update the training and testing sets with the filled data
train_filled, test_filled = weekly_demand_filled[0:split_point], weekly_demand_filled[split_point:]

# Using Auto ARIMA to find the best ARIMA model parameters would be ideal, but educated guesses
# were used for the ARIMA parameters based on typical practices and seasonality observed previously.
# p=1, d=1, q=1, is a common starting point.

# Step 4: Fit ARIMA Model
model = ARIMA(train_filled, order=(1, 1, 2))
model_fit = model.fit()

# Step 5: Make Predictions
predictions = model_fit.forecast(steps=len(test_filled))
predictions_indexed = pd.Series(predictions, index=test_filled.index)

# Step 6: Evaluate Model Performance
rmse = sqrt(mean_squared_error(test_filled, predictions))

print(rmse, predictions_indexed.head())
```

Figure 31. Code showing data preparation and use of ARIMA for forecasting weekly average demand rate.

The fitted ARIMA model predicted the weekly average demand for the testing period, generating forecasted values. Model performance was assessed using the Root Mean Square Error (RMSE), which quantified prediction accuracy. The achieved RMSE of approximately 0.283 indicated the model's predictive capability, with errors averaging around this value, reflecting on the model's effectiveness in forecasting weekly demand trends based on historical data.

```
0.2829874408920788 timestamp
2018-02-11    4.252575
2018-02-18    4.343323
2018-02-25    4.414891
2018-03-04    4.471332
2018-03-11    4.515844
Freq: W-SUN, Name: predicted_mean, dtype: float64
```

Figure 32. Output showing model performance and predictions.

Part 5

A comparative analysis was conducted using a Random Forest Regressor and a Deep Neural Network (DNN) to predict the demand rate, focusing on Mean Squared Error (MSE) as the primary evaluation metric.

The Random Forest Regressor, after being trained on a dataset with one-hot encoded categorical features and extracted time-related features from timestamps, achieved an MSE of approximately 0.197. The dataset was split into training (70%) and testing (30%) portions for this purpose.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
import numpy as np

# Step 1: Feature Selection and Preprocessing
# Extracting time-related features from 'timestamp' index
data_interpolated['hour'] = data_interpolated.index.hour
data_interpolated['dayofweek'] = data_interpolated.index.dayofweek

# Selecting relevant features and target variable
features = ['season', 'holiday', 'workingday', 'weather', 'temp', 'temp_feel', 'humidity', 'windspeed', 'hour', 'dayofweek']
X = data_interpolated[features]
y = data_interpolated['demand']

# Step 2: Encode Categorical Variables
# Identifying categorical columns for one-hot encoding
categorical_features = ['season', 'holiday', 'workingday', 'weather']
one_hot = OneHotEncoder()
transformer = ColumnTransformer([("one_hot", one_hot, categorical_features)], remainder="passthrough")

# Step 3: Split the Data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Preparing the Random Forest pipeline with preprocessing
rf_pipeline = Pipeline(steps=[('preprocessor', transformer),
                              ('model', RandomForestRegressor(n_estimators=100, random_state=42))])

# Training the Random Forest model
rf_pipeline.fit(X_train, y_train)

# Predicting with the Random Forest model
rf_predictions = rf_pipeline.predict(X_test)

# Evaluating the Random Forest model
rf_mse = mean_squared_error(y_test, rf_predictions)

print(rf_mse)
```

Figure 33. Code showing preparation of data and model training.

```
0.19693038189835663
```

Figure 34. Output showing mean square error for Random Forest Regressor.

Subsequently, a DNN was developed and trained on the same dataset. Preprocessing included scaling numerical features to a standard range to accommodate the neural network's input requirements. The DNN consisted of input, hidden, and output layers tailored for regression tasks and used a regression-appropriate loss function. The DNN achieved an MSE of approximately 0.183 on the test set.

```
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Data Preprocessing for DNN
# Scaling numerical features and applying OneHot encoding within a pipeline for DNN
numerical_features = ['temp', 'temp_feel', 'humidity', 'windspeed', 'hour', 'dayofweek']
preprocessor_for_dnn = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(), categorical_features)],
    remainder='passthrough')

# Apply preprocessing
X_train_preprocessed = preprocessor_for_dnn.fit_transform(X_train)
X_test_preprocessed = preprocessor_for_dnn.transform(X_test)

# Designing the DNN Architecture
model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_preprocessed.shape[1],)),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1) # Output layer for regression
])

# Compiling the DNN
model.compile(optimizer=Adam(learning_rate=0.001), loss='mean_squared_error')

# Training the DNN
model.fit(X_train_preprocessed, y_train, validation_split=0.2, epochs=100, batch_size=32, verbose=0)

# Predicting with the DNN
dnn_predictions = model.predict(X_test_preprocessed)

# Evaluating the DNN model
dnn_mse = mean_squared_error(y_test, dnn_predictions)

print(dnn_mse)
```

Figure 35. Code showing preprocessing of data and training of deep neural network.

```
 1/82 [..............................] - ETA: 4s
82/82 [==============================] - 0s 622us/step
0.18282145443501724
```

Figure 36. Output of deep neural network training and testing, showing mean squared error.

Performance Comparison:

Random Forest MSE: 0.196

DNN MSE: 0.183

The DNN slightly outperformed the Random Forest Regressor, indicating a marginally better capacity to predict demand rate. The minimal difference in MSE suggests the DNN could capture complex patterns in the data more effectively.

Reasons for DNN's Slight Superiority:

Complex Non-linearities Modelling: DNNs are adept at modelling complex non-linear relationships within data, likely contributing to their nuanced understanding of variables like weather conditions and time, which influence demand.

Feature Representation Learning: Unlike Random Forests, which rely on predefined features, DNNs can learn and combine features in predictive ways, potentially uncovering subtle data patterns.

Scalability: DNNs excel with large datasets, possibly continuing to improve as more data becomes available, in contrast to Random Forests, which might reach performance plateaus.

However, the performance gap between the models is slim, emphasizing the need to weigh additional factors like training time, computational resources, and model interpretability in practical applications. While DNNs offer complex modelling capabilities and scalability, Random Forests provide easier interpretation and require less data preprocessing, making them advantageous in specific contexts. Additionally, the complexity of DNNs increases the risk of overfitting, necessitating techniques like regularization to maintain model generalizability.

Part 6

For categorising demand rates into groups above and below the average demand rate, labelled 1 and 2 respectively, the following methodology was executed:

1. **Categorisation:** Demand rates were divided based on their comparison to the average demand rate.
2. **Data Preparation:** The dataset was pre-processed, making it suitable for classification.
3. **Data Splitting:** We allocated 70% of the data for training and 30% for testing.
4. **Model Training:** Three classifiers were trained: Logistic Regression, Random Forest Classifier, and Support Vector Machine (SVM).
5. **Evaluation:** Each model's accuracy was determined by predicting labels on the test set.

```python
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Step 1: Categorize Demand Rates
average_demand = data_interpolated['demand'].mean()
data_interpolated['demand_label'] = np.where(data_interpolated['demand'] > average_demand, 1, 2)

# Prepare features and target variable for classification
X = data_interpolated[features]  # Using the same features as before
y_class = data_interpolated['demand_label']

# Split the data
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(X, y_class, test_size=0.3, random_state=42)

# Data Preprocessing: Applying the same preprocessing (OneHot encoding for categorical features)
X_train_preprocessed_class = preprocessor_for_dnn.fit_transform(X_train_class)
X_test_preprocessed_class = preprocessor_for_dnn.transform(X_test_class)

# Initialize classifiers
classifiers = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(n_estimators=100, random_state=42),
    "SVM": SVC()
}

# Train, predict, and evaluate each classifier
accuracy_scores = {}
for name, clf in classifiers.items():
    clf.fit(X_train_preprocessed_class, y_train_class)
    predictions = clf.predict(X_test_preprocessed_class)
    accuracy = accuracy_score(y_test_class, predictions)
    accuracy_scores[name] = accuracy

print(accuracy_scores)
```

Figure 37. Code Showing training of three different classification models.

The performance of the classifiers is as follows:

- **Logistic Regression:** Showed an accuracy of approximately 79.03%, the least effective among the three, likely due to its linear nature which might not capture complex patterns in demand rates as efficiently.
- **Random Forest Classifier:** Achieved the highest accuracy at approximately 91.66%, indicating its superior capability in handling the task's non-linear relationships and feature interactions.
- **Support Vector Machine (SVM):** Recorded an accuracy of approximately 88.29%, demonstrating effective boundary detection between classes, though slightly less accurate than the Random Forest Classifier.

```
{'Logistic Regression': 0.7902793723689246, 'Random Forest': 0.9165709911978569, 'SVM':
 0.8828932261768083}
```

Figure 38. Output showing accuracy of each classifier model.

The Random Forest Classifier outperformed the others, likely due to its robustness in managing non-linear data and complex interactions, making it the most suitable model for this classification task. SVMs also showed commendable performance, benefitting from their ability to delineate complex class boundaries. The comparative lower performance of Logistic Regression underscores its limitations in handling the nuanced differentiation required for this specific classification challenge.

Part 7

To analyse the uniformity of clusters for temperature data in 2017, K-Means and Agglomerative Hierarchical Clustering methods were applied across different cluster sizes (k=2, 3, 4, and 12). The aim was to find which k value yields the most uniform clusters, defined by a close number of samples in each cluster.

Steps for Clustering:

1. Extracted the 2017 temperature data.
2. Applied both K-Means and Agglomerative Hierarchical Clustering for each k value.
3. Assessed uniformity by calculating the standard deviation of the number of samples in each cluster, with lower values indicating more uniform clusters.

```
from sklearn.cluster import KMeans, AgglomerativeClustering
import numpy as np

# Step 1: Prepare the Data
# Extracting temperature data for 2017
temp_data_2017 = data_interpolated[data_interpolated.index.year == 2017]['temp'].values.reshape(-1, 1)  # Reshape for clustering

# Define k values to be evaluated
k_values = [2, 3, 4, 12]

# Initialize dictionaries to store the results
kmeans_results = {}
agglomerative_results = {}

# Perform clustering for each k value using both methods
for k in k_values:
    # K-Means Clustering
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans_clusters = kmeans.fit_predict(temp_data_2017)
    kmeans_cluster_sizes = [list(kmeans_clusters).count(i) for i in range(k)]
    kmeans_results[k] = kmeans_cluster_sizes

    # Agglomerative Hierarchical Clustering
    agglomerative = AgglomerativeClustering(n_clusters=k)
    agglomerative_clusters = agglomerative.fit_predict(temp_data_2017)
    agglomerative_cluster_sizes = [list(agglomerative_clusters).count(i) for i in range(k)]
    agglomerative_results[k] = agglomerative_cluster_sizes

# Evaluate uniformity by calculating the standard deviation of cluster sizes for each k
kmeans_uniformity = {k: np.std(sizes) for k, sizes in kmeans_results.items()}
agglomerative_uniformity = {k: np.std(sizes) for k, sizes in agglomerative_results.items()}

print(kmeans_uniformity, agglomerative_uniformity)
```

Figure 39. Code for performing clustering and calculating uniformity using standard deviation.

```
{2: 22.0, 3: 23.01207412545761, 4: 206.89792169086667, 12: 139.35914832626605} {2: 61.0
, 3: 694.0995765898595, 4: 365.7741789683903, 12: 210.0185838338016}
```

Figure 40. Output showing results of clustering uniformity.

Clustering Results:

K-Means Clustering showed the following standard deviations for cluster sizes:

- k=2: 22.0
- k=3: 23.01
- k=4: 206.90
- k=12: 139.36

Agglomerative Hierarchical Clustering demonstrated:

- k=2: 61.0
- k=3: 694.10
- k=4: 365.77
- k=12: 210.02

K-Means Clustering with k=2 resulted in the most uniform clusters, evidenced by the lowest standard deviation. This suggests a balanced distribution of temperature samples when segmented into two broad categories. Agglomerative Hierarchical Clustering also indicated more uniform clusters at k=2, albeit with a higher standard deviation than K-Means, pointing to less uniformity in sample distribution.

For both clustering methods, k=2 yielded the most uniform clusters, indicating a natural division of the temperature data into two distinct groups provides a balanced sample distribution. K-Means Clustering outperformed Agglomerative Hierarchical Clustering in achieving more uniform clusters, especially notable at k=2. This analysis suggests that a simpler division of temperature data into two categories (e.g., higher vs. lower temperatures) results in a more even distribution of samples, with K-Means offering the most balanced and uniform clustering solution. Higher k values, particularly in Agglomerative Clustering, led to increased variability in cluster sizes, underscoring the challenge of maintaining uniformity with more granular clustering.

References

- Beaulieu, A. (2022) Learning SQL: Generate, Manipulate, and Retrieve Data. 3rd ed. O'Reilly Media.
- Nield, T. (2021) Getting Started with SQL: A Hands-On Approach for Beginners. O'Reilly Media.
- McKinney, W. (2018) Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. 2nd ed. O'Reilly Media.
- VanderPlas, J. (2016) Python Data Science Handbook: Essential Tools for Working with Data. O'Reilly Media.
- Hyndman, R.J. and Athanasopoulos, G. (2018) Forecasting: Principles and Practice. 3rd ed. OTexts.
- James, G., Witten, D., Hastie, T., and Tibshirani, R. (2021) An Introduction to Statistical Learning: with Applications in R. 2nd ed. Springer.
- Raschka, S. and Mirjalili, V. (2019) Python Machine Learning. 3rd ed. Packt Publishing.

Appendices

Appendix A: Code for Database Management tasks.

```
################################Part 1#####################################

'''
Create an SQLite database and import the data into a table named
“CarSharing”. Create a backup table and copy the whole table into it.
'''

import pandas as pd

# Load the CSV file to understand its structure
csv_file_path = 'CarSharing.csv'
car_sharing_df = pd.read_csv(csv_file_path)

# Display the first few rows of the dataframe to understand its structure and
data types
print(car_sharing_df.head(), car_sharing_df.dtypes)

import sqlite3

# Create a new SQLite database
db_path = 'CarSharingDB.db'
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Create the CarSharing table
create_table_query = '''
CREATE TABLE CarSharing (
    id INTEGER PRIMARY KEY,
    timestamp TEXT,
    season TEXT,
    holiday TEXT,
    workingday TEXT,
    weather TEXT,
    temp REAL,
    temp_feel REAL,
    humidity REAL,
    windspeed REAL,
    demand REAL
)
'''
cursor.execute(create_table_query)

# Import data into the CarSharing table
```

```
car_sharing_df.to_sql('CarSharing', conn, if_exists='replace', index=False)

# Create a backup table with the same structure as CarSharing
cursor.execute('CREATE TABLE CarSharingBackup AS SELECT * FROM CarSharing
WHERE 1=0')

# Copy all data from CarSharing to CarSharingBackup
cursor.execute('INSERT INTO CarSharingBackup SELECT * FROM CarSharing')

# Commit the changes and close the connection
conn.commit()

# Verify by counting the rows in both tables
row_count_carsharing = cursor.execute('SELECT COUNT(*) FROM
CarSharing').fetchone()[0]
row_count_backup = cursor.execute('SELECT COUNT(*) FROM
CarSharingBackup').fetchone()[0]

conn.close()

print(row_count_carsharing, row_count_backup)


########################################Part
2#####################################

'''
Add a column to the CarSharing table named “temp_category”. This
column should contain three string values. If the “feels-like”
temperature is less than 10 then the corresponding value in the
temp_category column should be “Cold”, if the feels-like temperature is
between 10 and 25, the value should be “Mild”, and if the feels-like
temperature is greater than 25, then the value should be “Hot”.
'''

# Reopen the connection to the SQLite database
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Step 1: Add the new column "temp_category" to the "CarSharing" table
cursor.execute('ALTER TABLE CarSharing ADD COLUMN temp_category TEXT')

# Step 2: Update the "temp_category" column based on "temp_feel" values
cursor.execute('UPDATE CarSharing SET temp_category = "Cold" WHERE temp_feel <
10')
cursor.execute('UPDATE CarSharing SET temp_category = "Mild" WHERE temp_feel
>= 10 AND temp_feel <= 25')
cursor.execute('UPDATE CarSharing SET temp_category = "Hot" WHERE temp_feel >
25')
```

```
# Commit the changes
conn.commit()

# Step 3: Verification - Select a few rows to check the "temp_category" values
verification_query = 'SELECT id, temp_feel, temp_category FROM CarSharing
LIMIT 10'
verification_results = cursor.execute(verification_query).fetchall()

conn.close()

print(verification_results)

##################################Part 3#####################################

'''
Create another table named “temperature” by selecting the temp,
temp_feel, and temp_category columns. Then drop the temp and temp_feel
columns from the CarSharing table.
'''

# Reopen the connection to the SQLite database
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Step 1: Create the "temperature" table
cursor.execute('CREATE TABLE temperature AS SELECT id, temp, temp_feel,
temp_category FROM CarSharing')

# Step 2: Remove the "temp" and "temp_feel" columns from the "CarSharing"
table
# 2.1 Create a temporary table excluding the "temp" and "temp_feel" columns
cursor.execute('''
CREATE TABLE CarSharing_temp AS
SELECT id, timestamp, season, holiday, workingday, weather, humidity,
windspeed, demand, temp_category
FROM CarSharing
''')

# 2.2 Drop the original "CarSharing" table
cursor.execute('DROP TABLE CarSharing')

# 2.3 Rename the temporary table to "CarSharing"
cursor.execute('ALTER TABLE CarSharing_temp RENAME TO CarSharing')

# Commit the changes
conn.commit()
```

```
# Step 3: Verification
# 3.1 Verify the "temperature" table structure
temperature_table_structure = cursor.execute('PRAGMA
table_info(temperature)').fetchall()

# 3.2 Verify the modified "CarSharing" table structure
carsharing_table_structure = cursor.execute('PRAGMA
table_info(CarSharing)').fetchall()

conn.close()

print(temperature_table_structure, carsharing_table_structure)

###################################Part 4#################################

'''
Find the distinct values of the weather column and assign a number
to each value. Add another column named “weather_code” to the table
containing each row’s assigned weather code.
'''

# Reopen the connection to the SQLite database
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Step 1: Identify distinct weather values
distinct_weather = cursor.execute('SELECT DISTINCT weather FROM
CarSharing').fetchall()
distinct_weather = [weather[0] for weather in distinct_weather]

# Step 2: Assign a unique number to each weather condition
weather_codes = {weather: code for code, weather in
enumerate(distinct_weather, start=1)}

# Step 3: Add the "weather_code" column to the "CarSharing" table
cursor.execute('ALTER TABLE CarSharing ADD COLUMN weather_code INTEGER')

# Step 4: Update the "weather_code" column based on the weather condition
for weather, code in weather_codes.items():
    cursor.execute('UPDATE CarSharing SET weather_code = ? WHERE weather = ?',
(code, weather))

# Commit the changes
conn.commit()

# Step 5: Verification - Select a few rows to check the "weather" and
"weather_code" values
```

```
verification_query = 'SELECT id, weather, weather_code FROM CarSharing LIMIT
10'
verification_results = cursor.execute(verification_query).fetchall()

conn.close()

print(weather_codes, verification_results)

###################################Part 5#####################################

'''
Create a table called “weather” and copy the columns “weather” and
“weather_code” to this table. Then drop the weather column from the
CarSharing table.
'''

# Reopen the connection to the SQLite database
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Step 1: Create the "weather" table with unique pairs of "weather" conditions
and "weather_code"
cursor.execute('CREATE TABLE weather AS SELECT DISTINCT weather, weather_code
FROM CarSharing')

# Step 3: Remove the "weather" column from the "CarSharing" table using the
workaround
# 3.1 Create a temporary table without the "weather" column
cursor.execute('''
CREATE TABLE CarSharing_temp AS
SELECT id, timestamp, season, holiday, workingday, humidity, windspeed,
demand, temp_category, weather_code
FROM CarSharing
''')

# 3.2 Drop the original "CarSharing" table
cursor.execute('DROP TABLE CarSharing')

# 3.3 Rename the temporary table to "CarSharing"
cursor.execute('ALTER TABLE CarSharing_temp RENAME TO CarSharing')

# Commit the changes
conn.commit()

# Step 4: Verification
# 4.1 Verify the "weather" table structure
weather_table_structure = cursor.execute('PRAGMA
table_info(weather)').fetchall()
```

```
# 4.2 Verify the modified "CarSharing" table structure
carsharing_table_structure = cursor.execute('PRAGMA
table_info(CarSharing)').fetchall()

conn.close()

print(weather_table_structure, carsharing_table_structure)

#####################################Part
6#####################################

'''
Create a table called "time" with four columns containing each row’s
timestamp, hour, weekday name, and month name (Hint: you can use the
strftime() function for this purpose).
'''

# Reopen the connection to the SQLite database
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Step 1 & 2: Create the "time" table with timestamp, hour, weekday name, and
month name
# SQLite treats Monday as 1, hence the adjustment in weekday calculation to
match common expectations
cursor.execute('''
CREATE TABLE time AS
SELECT
    timestamp,
    CAST(strftime('%H', timestamp) AS INTEGER) AS hour,
    CASE CAST(strftime('%w', timestamp) AS INTEGER)
        WHEN 0 THEN 'Sunday'
        WHEN 1 THEN 'Monday'
        WHEN 2 THEN 'Tuesday'
        WHEN 3 THEN 'Wednesday'
        WHEN 4 THEN 'Thursday'
        WHEN 5 THEN 'Friday'
        WHEN 6 THEN 'Saturday'
    END AS weekday,
    CASE CAST(strftime('%m', timestamp) AS INTEGER)
        WHEN 1 THEN 'January'
        WHEN 2 THEN 'February'
        WHEN 3 THEN 'March'
        WHEN 4 THEN 'April'
        WHEN 5 THEN 'May'
        WHEN 6 THEN 'June'
        WHEN 7 THEN 'July'
```

```
        WHEN 8 THEN 'August'
        WHEN 9 THEN 'September'
        WHEN 10 THEN 'October'
        WHEN 11 THEN 'November'
        WHEN 12 THEN 'December'
    END AS month
FROM CarSharing
''')

# Commit the changes
conn.commit()

# Step 4: Verification - Select a few rows to check the "time" table structure
and data
verification_query = 'SELECT * FROM time LIMIT 10'
verification_results = cursor.execute(verification_query).fetchall()

conn.close()

print(verification_results)


#####################################Part 7#################################

'''
Assume it’s the first day you have started working at this company
and your boss Linda sends you an email as follows: “Hello, welcome to
the team. I hope you enjoy working at this company. Could you please
give me a report containing the following information:

a) Please tell me which date and time we had the highest demand rate in 2017.

b) Give me a table containing the name of the weekday, month, and season
in which we had the highest and lowest average demand rates throughout
2017. Please include the calculated average demand values as well.

c) For the weekday selected in (b), please give me a table showing the
average demand rate we had at different hours of that weekday throughout
2017. Please sort the results in descending order based on the average
demand rates.

d) Please tell me what the weather was like in 2017. Was it mostly cold,
mild, or hot? Which weather condition (shown in the weather column) was
the most prevalent in 2017? What was the average, highest, and lowest
wind speed and humidity for each month in 2017? Please organise this
information in two tables for the wind speed and humidity. Please also
give me a table showing the average demand rate for each cold, mild, and
hot weather in 2017 sorted in descending order based on their average
demand rates.
```

```
e) Give me another table showing the information requested in (d) for
the month we had the highest average demand rate in 2017 so that I can
compare it with other months.
'''

# a

# Reopen the connection to the SQLite database
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Part (a): Highest demand rate date and time in 2017
highest_demand_query = '''
SELECT timestamp, MAX(demand) as max_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
'''
highest_demand_result = cursor.execute(highest_demand_query).fetchone()

print(highest_demand_result)

# b

# Part (b): Weekday, Month, and Season with Highest and Lowest Average Demand
Rates in 2017
average_demand_query = '''
SELECT t.weekday, t.month, c.season, AVG(c.demand) as avg_demand
FROM CarSharing c
JOIN time t ON c.timestamp = t.timestamp
WHERE strftime('%Y', c.timestamp) = '2017'
GROUP BY t.weekday, t.month, c.season
ORDER BY avg_demand DESC
'''

average_demand_results = cursor.execute(average_demand_query).fetchall()

# Extracting the rows with the highest and lowest average demand
highest_average_demand = average_demand_results[0]
lowest_average_demand = average_demand_results[-1]

print(highest_average_demand, lowest_average_demand)

# c

# Part (c): Average Demand Rate at Different Hours for Sunday throughout 2017
average_demand_by_hour_query = '''
SELECT t.hour, AVG(c.demand) as avg_demand
```

```
FROM CarSharing c
JOIN time t ON c.timestamp = t.timestamp
WHERE strftime('%Y', c.timestamp) = '2017' AND t.weekday = 'Sunday'
GROUP BY t.hour
ORDER BY avg_demand DESC
'''

average_demand_by_hour_results =
cursor.execute(average_demand_by_hour_query).fetchall()

print(average_demand_by_hour_results)

# d

# Part (d) Weather overview in 2017: Predominant temperature category and
prevalent weather condition
temperature_category_query = '''
SELECT temp_category, COUNT(temp_category) as count
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
GROUP BY temp_category
ORDER BY count DESC
LIMIT 1
'''

most_prevalent_weather_condition_query = '''
SELECT weather, COUNT(weather) as count
FROM weather
JOIN CarSharing ON weather.weather_code = CarSharing.weather_code
WHERE strftime('%Y', CarSharing.timestamp) = '2017'
GROUP BY weather
ORDER BY count DESC
LIMIT 1
'''

# Average, highest, and lowest wind speed and humidity for each month in 2017
wind_speed_humidity_query = '''
SELECT
    strftime('%m', timestamp) as month,
    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
ORDER BY month
'''
```

```
# Average demand rate for each weather condition in 2017
average_demand_by_weather_condition_query = '''
SELECT temp_category, AVG(demand) as avg_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
GROUP BY temp_category
ORDER BY avg_demand DESC
'''

# Execute queries
predominant_temperature_category =
cursor.execute(temperature_category_query).fetchone()
most_prevalent_weather_condition =
cursor.execute(most_prevalent_weather_condition_query).fetchone()
wind_speed_humidity_stats =
cursor.execute(wind_speed_humidity_query).fetchall()
average_demand_by_weather_condition =
cursor.execute(average_demand_by_weather_condition_query).fetchall()

print(predominant_temperature_category)
print(most_prevalent_weather_condition)
print(wind_speed_humidity_stats)
print(average_demand_by_weather_condition)

# e

# Reopen the connection to the SQLite database for the targeted analysis
conn = sqlite3.connect(db_path)
cursor = conn.cursor()

# Step 1: Identify the month with the highest average demand rate in 2017
highest_avg_demand_month_query = '''
SELECT strftime('%m', timestamp) AS month, AVG(demand) AS avg_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017'
GROUP BY month
ORDER BY avg_demand DESC
LIMIT 1
'''
highest_avg_demand_month =
cursor.execute(highest_avg_demand_month_query).fetchone()[0]

# Step 2: Retrieve wind speed and humidity statistics for the identified month
wind_speed_humidity_for_month_query = '''
SELECT
    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' AND strftime('%m', timestamp) = ?
'''
wind_speed_humidity_for_month_stats =
cursor.execute(wind_speed_humidity_for_month_query,
(highest_avg_demand_month,)).fetchall()

# Step 3: Compare average demand rates by temperature category for the
identified month
average_demand_by_temp_category_for_month_query = '''
SELECT temp_category, AVG(demand) AS avg_demand
FROM CarSharing
WHERE strftime('%Y', timestamp) = '2017' AND strftime('%m', timestamp) = ?
GROUP BY temp_category
ORDER BY avg_demand DESC
'''
average_demand_by_temp_category_for_month =
cursor.execute(average_demand_by_temp_category_for_month_query,
(highest_avg_demand_month,)).fetchall()

# Close the connection
conn.close()

print(highest_avg_demand_month, wind_speed_humidity_for_month_stats,
average_demand_by_temp_category_for_month)
```

Appendix B: Code for Data Analytics tasks.

```
# First, let's load the data to understand its structure, and to identify any
duplicates and null values.
import pandas as pd

# Load the CSV file
file_path = 'CarSharing.csv'
data = pd.read_csv(file_path)

# Display the first few rows of the dataset to understand its structure
print(data.head())

###################################Part 1#####################################

# Step 1: Drop duplicate rows
data_deduplicated = data.drop_duplicates()

# Check the shape of the original vs deduplicated data to understand how many
duplicates were removed
original_shape = data.shape
deduplicated_shape = data_deduplicated.shape

# Step 2: Identify columns with null values and their counts
null_values_count = data_deduplicated.isnull().sum()

print(original_shape, deduplicated_shape, null_values_count)

# Analyzing the distributions of the columns with null values to decide on the
best imputation method
import matplotlib.pyplot as plt

# Plot histograms for each column with null values
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
columns_with_nulls = ['temp', 'temp_feel', 'humidity', 'windspeed']

for col, ax in zip(columns_with_nulls, axes.flatten()):
    data_deduplicated[col].hist(ax=ax, bins=30, alpha=0.7, color='blue')
    ax.set_title(col)

plt.tight_layout()
plt.show()

# Interpolating null values for 'temp', 'temp_feel', 'humidity', and
'windspeed'
data_interpolated = data_deduplicated.copy()
data_interpolated[['temp', 'temp_feel', 'humidity', 'windspeed']] =
data_interpolated[['temp', 'temp_feel', 'humidity',
'windspeed']].interpolate(method='linear')
```

```
# Verify if there are any null values left
remaining_nulls_after_interpolation = data_interpolated.isnull().sum()

print(remaining_nulls_after_interpolation)

###################################Part 2#####################################

from scipy.stats import pearsonr, f_oneway

# Identifying numerical and categorical columns
numerical_columns = data_interpolated.select_dtypes(include=['float64',
'int64']).columns.drop(['id', 'demand'])
categorical_columns =
data_interpolated.select_dtypes(include=['object']).columns.drop(['timestamp']
)

# Initializing dictionaries to store test results
pearson_results = {}
anova_results = {}

# Pearson Correlation for Numerical Columns
for col in numerical_columns:
    corr, p_value = pearsonr(data_interpolated[col],
data_interpolated['demand'])
    pearson_results[col] = (corr, p_value)

# ANOVA for Categorical Columns
for col in categorical_columns:
    groups = [data_interpolated['demand'][data_interpolated[col] == category]
for category in data_interpolated[col].unique()]
    f_stat, p_value = f_oneway(*groups)
    anova_results[col] = (f_stat, p_value)

print(pearson_results, anova_results)

###################################Part 3#####################################
import numpy as np

# Step 1: Convert 'timestamp' to datetime format
data_interpolated['timestamp'] =
pd.to_datetime(data_interpolated['timestamp'])
#modification

# Step 2: Set 'timestamp' as the index of the DataFrame
data_interpolated.set_index('timestamp', inplace=True)
```

```
# Step 3: Extract Data for 2017 (the dataset is already for 2017 based on
previous information, so this step is to ensure consistency)
data_2017 = data_interpolated[data_interpolated.index.year == 2017]
#modification

# Select only numeric columns before resampling
numeric_cols = data_2017.select_dtypes(include=[np.number])  # Assuming you
have imported NumPy as np

# Then resample and calculate the mean
data_monthly = numeric_cols.resample('M').mean()

'''
# Step 4: Resample Data Monthly to observe trends
data_monthly = data_2017.resample('M').mean()
'''

# Step 5: Plotting the resampled data to identify seasonal or cyclic patterns

fig, axes = plt.subplots(4, 1, figsize=(12, 16))
variables_to_plot = ['temp', 'humidity', 'windspeed', 'demand']

for var, ax in zip(variables_to_plot, axes):
    data_monthly[var].plot(ax=ax, title=var)
    ax.set_ylabel(var)
    ax.grid(True)

plt.tight_layout()
plt.show()

###################################Part 4#################################

from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

# Step 1: Prepare Dataset - Convert demand data into weekly averages
weekly_demand = data_interpolated['demand'].resample('W').mean()

# Step 2: Split Dataset into training and testing sets
split_point = int(len(weekly_demand) * 0.7)
train, test = weekly_demand[0:split_point], weekly_demand[split_point:]

# Display the first few entries of the weekly average demand to verify the
transformation
print(weekly_demand.head(), train.shape, test.shape)
```

```
# Handling NaN value in weekly_demand by filling it with the mean of the
series
weekly_demand_filled = weekly_demand.fillna(weekly_demand.mean())

# Update the training and testing sets with the filled data
train_filled, test_filled = weekly_demand_filled[0:split_point],
weekly_demand_filled[split_point:]

# Using Auto ARIMA to find the best ARIMA model parameters would be ideal, but
educated guesses
# were used for the ARIMA parameters based on typical practices and
seasonality observed previously.
# p=1, d=1, q=1, is a common starting point.

# Step 4: Fit ARIMA Model
model = ARIMA(train_filled, order=(1, 1, 2))
model_fit = model.fit()

# Step 5: Make Predictions
predictions = model_fit.forecast(steps=len(test_filled))
predictions_indexed = pd.Series(predictions, index=test_filled.index)

# Step 6: Evaluate Model Performance
rmse = sqrt(mean_squared_error(test_filled, predictions))

print(rmse, predictions_indexed.head())

###################################Part 5#####################################

#Random Forest Regressor

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
import numpy as np

# Step 1: Feature Selection and Preprocessing
# Extracting time-related features from 'timestamp' index
data_interpolated['hour'] = data_interpolated.index.hour
data_interpolated['dayofweek'] = data_interpolated.index.dayofweek

# Selecting relevant features and target variable
features = ['season', 'holiday', 'workingday', 'weather', 'temp', 'temp_feel',
'humidity', 'windspeed', 'hour', 'dayofweek']
X = data_interpolated[features]
```

```
y = data_interpolated['demand']

# Step 2: Encode Categorical Variables
# Identifying categorical columns for one-hot encoding
categorical_features = ['season', 'holiday', 'workingday', 'weather']
one_hot = OneHotEncoder()
transformer = ColumnTransformer([("one_hot", one_hot, categorical_features)],
remainder="passthrough")

# Step 3: Split the Data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Preparing the Random Forest pipeline with preprocessing
rf_pipeline = Pipeline(steps=[('preprocessor', transformer),
                              ('model',
RandomForestRegressor(n_estimators=100, random_state=42))])

# Training the Random Forest model
rf_pipeline.fit(X_train, y_train)

# Predicting with the Random Forest model
rf_predictions = rf_pipeline.predict(X_test)

# Evaluating the Random Forest model
rf_mse = mean_squared_error(y_test, rf_predictions)

print(rf_mse)

#Deep Neural Network

from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Data Preprocessing for DNN
# Scaling numerical features and applying OneHot encoding within a pipeline
for DNN
numerical_features = ['temp', 'temp_feel', 'humidity', 'windspeed', 'hour',
'dayofweek']
preprocessor_for_dnn = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(), categorical_features)],
    remainder='passthrough')

# Apply preprocessing
```

```
X_train_preprocessed = preprocessor_for_dnn.fit_transform(X_train)
X_test_preprocessed = preprocessor_for_dnn.transform(X_test)

# Designing the DNN Architecture
model = Sequential([
    Dense(128, activation='relu',
input_shape=(X_train_preprocessed.shape[1],)),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1) # Output layer for regression
])

# Compiling the DNN
model.compile(optimizer=Adam(learning_rate=0.001), loss='mean_squared_error')

# Training the DNN
model.fit(X_train_preprocessed, y_train, validation_split=0.2, epochs=100,
batch_size=32, verbose=0)

# Predicting with the DNN
dnn_predictions = model.predict(X_test_preprocessed)

# Evaluating the DNN model
dnn_mse = mean_squared_error(y_test, dnn_predictions)

print(dnn_mse)

#####################################Part
6#########################################

from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Step 1: Categorize Demand Rates
average_demand = data_interpolated['demand'].mean()
data_interpolated['demand_label'] = np.where(data_interpolated['demand'] >
average_demand, 1, 2)

# Prepare features and target variable for classification
X = data_interpolated[features]  # Using the same features as before
y_class = data_interpolated['demand_label']

# Split the data
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(X,
y_class, test_size=0.3, random_state=42)
```

```
# Data Preprocessing: Applying the same preprocessing (OneHot encoding for
categorical features)
X_train_preprocessed_class = preprocessor_for_dnn.fit_transform(X_train_class)
X_test_preprocessed_class = preprocessor_for_dnn.transform(X_test_class)

# Initialize classifiers
classifiers = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(n_estimators=100,
random_state=42),
    "SVM": SVC()
}

# Train, predict, and evaluate each classifier
accuracy_scores = {}
for name, clf in classifiers.items():
    clf.fit(X_train_preprocessed_class, y_train_class)
    predictions = clf.predict(X_test_preprocessed_class)
    accuracy = accuracy_score(y_test_class, predictions)
    accuracy_scores[name] = accuracy

print(accuracy_scores)

##############################################Part
7##############################################

from sklearn.cluster import KMeans, AgglomerativeClustering
import numpy as np

# Step 1: Prepare the Data
# Extracting temperature data for 2017
temp_data_2017 = data_interpolated[data_interpolated.index.year ==
2017]['temp'].values.reshape(-1, 1)  # Reshape for clustering

# Define k values to be evaluated
k_values = [2, 3, 4, 12]

# Initialize dictionaries to store the results
kmeans_results = {}
agglomerative_results = {}

# Perform clustering for each k value using both methods
for k in k_values:
    # K-Means Clustering
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans_clusters = kmeans.fit_predict(temp_data_2017)
    kmeans_cluster_sizes = [list(kmeans_clusters).count(i) for i in range(k)]
    kmeans_results[k] = kmeans_cluster_sizes
```

```
    # Agglomerative Hierarchical Clustering
    agglomerative = AgglomerativeClustering(n_clusters=k)
    agglomerative_clusters = agglomerative.fit_predict(temp_data_2017)
    agglomerative_cluster_sizes = [list(agglomerative_clusters).count(i) for i
in range(k)]
    agglomerative_results[k] = agglomerative_cluster_sizes

# Evaluate uniformity by calculating the standard deviation of cluster sizes
for each k
kmeans_uniformity = {k: np.std(sizes) for k, sizes in kmeans_results.items()}
agglomerative_uniformity = {k: np.std(sizes) for k, sizes in
agglomerative_results.items()}

print(kmeans_uniformity, agglomerative_uniformity)
```