########################## APPENDIX ##########################

############################ PART I ############################

Task 1

print("I am task 1 solution")

def connect_database(database_file):

```
conn = sqlite3.connect(database_file)
cursor = conn.cursor()
return conn, cursor
```

def close_database(conn):

```
conn.commit()
conn.close()
```

def import_dataset(cursor):

```
with open('CarSharing.csv', 'r') as file:
    next(file)  # Skip the header row
    for line in file:
        values = line.strip().split(',')
        cursor.execute('''
            INSERT INTO CarSharing (id, timestamp, season, holiday, workingday, weather, temp, temp_feel, humidity, windspeed, deman
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        ''', values)
```

def create_backup1_table(cursor):

```
cursor.execute('''
    CREATE TABLE Backup1 AS
    SELECT id, timestamp, season, holiday, workingday
    FROM CarSharing
    ORDER BY RANDOM()
    LIMIT (SELECT COUNT(*) FROM CarSharing) / 2
''')
```

def create_backup2_table(cursor):

```
cursor.execute('''
    CREATE TABLE Backup2 AS
    SELECT id, weather, temp, temp_feel, humidity, windspeed, demand
    FROM CarSharing
    WHERE id NOT IN (SELECT id FROM Backup1)
''')
```

Task 2

```
print("I am task 2 solution")
```

def add_humidity_category_col(cursor):

```
cursor.execute('ALTER TABLE CarSharing ADD COLUMN humidity_category TEXT')
cursor.execute('PRAGMA table_info(CarSharing)')
columns = cursor.fetchall()
column_names = [column[1] for column in columns]
if 'humidity_category' in column_names:
    print("The 'humidity_category' column already exists.")
    return
cursor.execute('''
    UPDATE CarSharing
    SET humidity_category =
        CASE
            WHEN humidity <= 55 THEN 'Dry'
            WHEN humidity > 55 AND humidity <= 65 THEN 'Sticky'
            WHEN humidity > 65 THEN 'Oppressive'
        END
''')
```

Task 3

```
print("I am task 3 solution")
```

3 (a)

def create_weather_table(cursor):

```
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Weather (
        id INTEGER PRIMARY KEY,
        weather INTEGER,
        temp REAL,
        temp_feel REAL,
        humidity INTEGER,
        windspeed REAL,
        humidity_category TEXT
    )
''')

# Copy the required columns from "CarSharing" to "Weather" table
cursor.execute('''
    INSERT INTO Weather (id, weather, temp, temp_feel, humidity, windspeed, humidity_category)
    SELECT id, weather, temp, temp_feel, humidity, windspeed, humidity_category
    FROM CarSharing
''')

# Get the column names from the CarSharing table
cursor.execute("PRAGMA table_info('CarSharing')")
columns = [column[1] for column in cursor.fetchall()]

# Generate the DROP statements for columns other than 'id' and 'demand'
drop_statements = [
    f"ALTER TABLE CarSharing DROP COLUMN {column}"
    for column in columns if column not in ['id', 'demand']
]

# Execute the DROP statements
for drop_statement in drop_statements:
    cursor.execute(drop_statement)

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

3 (b)

def assign_code_to_values(cursor):

```
 cursor.execute('SELECT DISTINCT workingday FROM CarSharing')
workingday_values = [row[0] for row in cursor.fetchall()]
cursor.execute('SELECT DISTINCT holiday FROM CarSharing')
holiday_values = [row[0] for row in cursor.fetchall()]
workingday_codes = {value: code for code, value in enumerate(workingday_values, start=1)}
holiday_codes = {value: code for code, value in enumerate(holiday_values, start=1)}
cursor.execute('PRAGMA table_info(CarSharing)')
columns = [row[1] for row in cursor.fetchall()]
if 'workingday_code' not in columns:
    cursor.execute('ALTER TABLE CarSharing ADD COLUMN workingday_code INTEGER')

if 'holiday_code' not in columns:
    cursor.execute('ALTER TABLE CarSharing ADD COLUMN holiday_code INTEGER')
for value in workingday_values:
    cursor.execute('UPDATE CarSharing SET workingday_code = ? WHERE workingday = ?', (workingday_codes[value], value))
for value in holiday_values:
    cursor.execute('UPDATE CarSharing SET holiday_code = ? WHERE holiday = ?', (holiday_codes[value], value))
conn.commit()
```

3(c)

def create_holiday_table(cursor):

```
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Holiday (
        id INTEGER PRIMARY KEY,
        holiday INTEGER,
        workingday INTEGER,
        workingday_code INTEGER,
        holiday_code INTEGER
    )
''')
cursor.execute('''
    INSERT INTO Holiday (id, holiday, workingday, workingday_code, holiday_code)
    SELECT id, holiday, workingday, workingday_code, holiday_code
    FROM CarSharing
''')
cursor.execute('''
    CREATE TABLE CarSharingNew AS
    SELECT id, season, timestamp, weather, temp, temp_feel, humidity, windspeed, demand
    FROM CarSharing
''')
cursor.execute('DROP TABLE CarSharing')
cursor.execute('ALTER TABLE CarSharingNew RENAME TO CarSharing')
conn.commit()
```

3(d)

def create_time_table(cursor):

```
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Time (
        id INTEGER PRIMARY KEY,
        timestamp TEXT,
        hour INTEGER,
        weekday_name TEXT,
        month TEXT,
        season_name TEXT
    )
''')
cursor.execute('''
    INSERT INTO Time (id, timestamp, hour, weekday_name, month, season_name)
    SELECT id,
           strftime('%Y-%m-%d %H:%M:%S', timestamp) AS timestamp,
           strftime('%H', timestamp) AS hour,
           strftime('%w', timestamp) AS weekday_name,
           strftime('%m', timestamp) AS month,
           CASE
               WHEN season = 1 THEN 'Spring'
               WHEN season = 2 THEN 'Summer'
               WHEN season = 3 THEN 'Fall'
               WHEN season = 4 THEN 'Winter'
           END AS season_name
    FROM CarSharing
''')
cursor.execute('''
    CREATE TABLE CarSharingNew AS
    SELECT id, weather, temp, temp_feel, humidity, windspeed, demand
    FROM CarSharing
''')
cursor.execute('DROP TABLE CarSharing')
cursor.execute('ALTER TABLE CarSharingNew RENAME TO CarSharing')
conn.commit()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Weather (
        id INTEGER PRIMARY KEY,
        weather INTEGER,
        temp REAL,
        temp_feel REAL,
        humidity INTEGER,
        windspeed REAL,
        humidity_category TEXT
    )
''')
cursor.execute('''
    INSERT INTO Weather (id, weather, temp, temp_feel, humidity, windspeed, humidity_category)
    SELECT id, weather, temp, temp_feel, humidity, windspeed, humidity_category
    FROM CarSharing
''')
cursor.execute("PRAGMA table_info('CarSharing')")
columns = [column[1] for column in cursor.fetchall()]
drop_statements = [
    f"ALTER TABLE CarSharing DROP COLUMN {column}"
    for column in columns if column not in ['id', 'demand']
]
for drop_statement in drop_statements:
    cursor.execute(drop_statement)
conn.commit()
```

Task 4

```
print("I am Task 4 Solution")
```

Query_4A

```
= '''
SELECT Time.timestamp, CarSharing.demand
FROM Time
JOIN CarSharing ON Time.id = CarSharing.id
JOIN Weather ON Weather.id = CarSharing.id
WHERE Weather.temp = (SELECT MIN(temp) FROM Weather)
'''
```

Query_4 B

```
= '''
SELECT h.workingday,
       AVG(w.windspeed) AS average_windspeed,
       MAX(w.windspeed) AS highest_windspeed,
       MIN(w.windspeed) AS lowest_windspeed,
       AVG(w.humidity) AS average_humidity,
       MAX(w.humidity) AS highest_humidity,
       MIN(w.humidity) AS lowest_humidity
FROM Weather AS w
JOIN Time AS t ON w.id = t.id
JOIN Holiday AS h ON w.id = h.id
WHERE h.workingday = 'Yes' OR h.workingday = 'No'
    AND t.timestamp LIKE '2017-%'
GROUP BY h.workingday
'''
```

Query_4C

```
='''
SELECT
Time.weekday_name,
Time.month,
Time.season_name,
AVG(CarSharing.demand) AS average_demand
FROM
Time
JOIN CarSharing ON Time.id = CarSharing.id
WHERE
strftime('%Y', Time.timestamp) = '2017'
GROUP BY
Time.weekday_name, Time.month, Time.season_name
HAVING
AVG(CarSharing.demand) = (
    SELECT
        MAX(avg_demand)
    FROM
        (SELECT
            AVG(CarSharing.demand) AS avg_demand
        FROM
            Time
            JOIN CarSharing ON Time.id = CarSharing.id
        WHERE
            strftime('%Y', Time.timestamp) = '2017'
        GROUP BY
            Time.weekday_name, Time.month, Time.season_name)
    )
    '''
```

Query_4D

```
= '''
SELECT Weather.humidity_category, AVG(CarSharing.demand) AS average_demand
FROM CarSharing
JOIN Time ON CarSharing.id = Time.id
JOIN Weather ON CarSharing.id = Weather.id
WHERE strftime('%Y', Time.timestamp) = '2017'
GROUP BY Weather.humidity_category
ORDER BY average_demand DESC
'''
```

########################## PART 2 ############################

Task 1

```
print("I am Task 1 Solution")
database_file = 'CarDatabase_dataAnalytics.db'
conn = sqlite3.connect(database_file)
df = pd.read_sql_query("SELECT * FROM CarSharing", conn)
print("Drop Dublicating.....")
df.drop_duplicates(inplace=True)
df.dropna(inplace=True)
```

Task 2

```
print("I am Task 2 Solution")
df['temp'] = pd.to_numeric(df['temp'], errors='coerce')
df['demand'] = pd.to_numeric(df['demand'], errors='coerce')
df['humidity'] = pd.to_numeric(df['humidity'], errors='coerce')
df['windspeed'] = pd.to_numeric(df['windspeed'], errors='coerce')
data1 = df[['temp','humidity','windspeed','demand']]
data1 = data1.dropna()
df.dropna(inplace=True)
corr_temp, p_temp = pearsonr(df['temp'], df['demand'])
corr_humidity, p_humidity = pearsonr(df['humidity'], df['demand'])
corr_windspeed, p_windspeed = pearsonr(df['windspeed'], df['demand'])
print("Correlation and p-values:")
print("Temperature vs. Demand: Correlation =", corr_temp, "p-value =", p_temp)
print("Humidity vs. Demand: Correlation =", corr_humidity, "p-value =", p_humidity)
print("Windspeed vs. Demand: Correlation =", corr_windspeed, "p-value =", p_windspeed)
```

T_Test

```
workingday = df[df['workingday'] == "Yes"]['demand']
non_workingday = df[df['workingday'] == "No"]['demand']
t_statistic, p_value = ttest_ind(workingday, non_workingday)
print(t_statistic)
print("T-test (Workingday vs. Non-workingday):")
print("t-statistic =", t_statistic, "p-value =", p_value)
```

Task 3

```
print("I am Task 3 Solution")
df['timestamp'] = pd.to_datetime(df['timestamp'])
df_2017 = df[df['timestamp'].dt.year == 2017]
df_weekly = df_2017.resample('W', on='timestamp')['temp'].mean().reset_index()
df_weekly['temp'].interpolate(method='linear', inplace=True)
train_data = df_weekly.iloc[:int(len(df_weekly) * 0.7)]
test_data = df_weekly.iloc[int(len(df_weekly) * 0.7):]
model = ARIMA(train_data['temp'], order=(1, 0, 0))
model_fit = model.fit()
predictions = model_fit.forecast(steps=len(test_data))
predictions_df = pd.DataFrame(predictions, columns=['predicted_temp'])
result_df = pd.concat([test_data.reset_index(drop=True), predictions_df], axis=1)
plt.plot(result_df['timestamp'], result_df['temp'], label='Actual')
plt.plot(result_df['timestamp'], predictions, label='Predicted')
plt.xlabel('Week')
plt.ylabel('Temperature')
plt.title('ARIMA Model - Weekly Average Temperature Prediction')
plt.legend()
plt.show()
```

Task 4

```
print("I am Task 4 Solution")
df.dropna(inplace=True)
X = df[['temp', 'humidity', 'windspeed', 'season']]
X['encoded_season'] = label_encoder.fit_transform(X['season'])
X.drop("season",inplace=True,axis=1)
y = df['weather']
encoded_y = label_encoder.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, encoded_y, test_size=0.3, random_state=42)
```

MODELS

1. Random

```
model3 = RandomForestClassifier()
model3.fit(X_train, y_train)
y_pred3 = model3.predict(X_test)
accuracy3 = accuracy_score(y_test, y_pred3)
print("Accuracy (Random Forest):", accuracy3)
```

2. DecsionTree

```
model2 = DecisionTreeClassifier()
model2.fit(X_train, y_train)
y_pred2 = model2.predict(X_test)
accuracy2 = accuracy_score(y_test, y_pred2)
print("Accuracy (Decision Tree):", accuracy2)
```

3. SVM Classifier

```
classifier = svm.SVC(kernel='linear')
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy (SVM classifier):", accuracy)
```

Task 5

```
print("I am Task 5 Solution")
model_nn = MLPRegressor(max_iter=1000)
param_grid = {
'hidden_layer_sizes': [(10,), (20,), (30,), (40,), (50,)],
'activation': ['relu', 'tanh'],
'solver': ['adam', 'sgd']
 }
grid_search = GridSearchCV(model_nn, param_grid, cv=5)
grid_search.fit(X_train, y_train)
best_model_nn = grid_search.best_estimator_
best_model_nn.fit(X_train, y_train)
predictions_nn = best_model_nn.predict(X_test)
mse_nn = mean_squared_error(y_test, predictions_nn)
print("Neural Network Mean Squared Error:", mse_nn)
model_rf = RandomForestRegressor()
param_grid = {
'n_estimators': [100, 200, 300],
'max_depth': [None, 5, 10],
'min_samples_split': [2, 5, 10]
 }
grid_search = GridSearchCV(model_rf, param_grid, cv=5)
grid_search.fit(X_train, y_train)
best_model_rf = grid_search.best_estimator_
print("Best Model ",best_model_rf)
best_model_rf.fit(X_train, y_train)
predictions_rf = best_model_rf.predict(X_test)
mse_rf = mean_squared_error(y_test, predictions_rf)
print("Random Forest Mean Squared Error:", mse_rf)
```

Task 6

```
print("I am Task 6 Solution")
scaler = StandardScaler()
humidity_data = df['humidity'].values.reshape(-1, 1)
humidity_data_scaled = scaler.fit_transform(humidity_data)

sse = []
K = range(1, 10)
for k in K:
  kmeans = KMeans(n_clusters=k, random_state=42)
  kmeans.fit(humidity_data_scaled)
  sse.append(kmeans.inertia_)

plt.plot(K, sse, 'bx-')
plt.xlabel('Number of Clusters')
plt.ylabel('Sum of Squared Distances')
plt.title('Elbow Method')
plt.show()
```