DTSC 691 Spring 1 2024

Capstone Project (Applied Data Science)

Paul J. Walker

Project Documentation

Table of Contents

Preface - PLEASE READ

My entire project is based on artificially generated data pertaining to used vehicle information due to limitations with obtaining real data. I am including this here in the preface because based on this data a lot of my analysis and/or visualizations do not make sense based on real-world logic. For example, when I ran an analysis on the effect of a vehicle's current condition on its sales price I found no correlation. In real life this does not hold true, as it is pretty obvious that the better the condition of the vehicle, the higher the price. I wanted to explain before the main content of my project in order to provide context for why things may seem inconsistent with preconceived notions with respect to used vehicle sales information.

Thank you for taking the time to examine my project.

1. Background Information

A. DBMS and its Functions

1. A database is an organized collection of structured data, typically stored and accessed electronically from a computer system. It consists of tables that contain rows and columns, with each row representing a record and each column representing a specific attribute or field.The most widely used type of database is a relational database. In a relational database data is organized into tables with predefined relationships between them, using SQL (Structured Query Language) for data manipulation and querying.

2. Databases are used by a large majority of business across various industries in the world and they have a wide range of applications such as;
   i. Data Storage and Organization: Businesses use databases to store vast amounts of data, including customer information, product details, sales transactions, inventory records, and more.They also enable organizations to organize data into logical structures

   ii. Data Analysis and Decision Making: Databases serve as a foundation for data analysis and business intelligence (BI) initiatives, providing a centralized repository for data used in reporting, dashboards, and analytics.

   iii. Customer Relationship Management (CRM): CRM systems rely on databases to store and manage customer data, including contact information, interactions, purchase history, preferences, and feedback.

   iv. Inventory Management and Supply Chain Optimization: Databases play a critical role in inventory management systems, tracking product quantities, locations, movements, and replenishment needs.

B. Availability of Used Vehicle Information

1. Data, regardless of industry, can be gathered from various sources both internally and externally.
    i. Internal Sources: Data generated or collected by the organization itself, including transactional data, customer records, sales reports, employee information, and operational metrics.
    ii. External Sources: Data obtained from third-party sources, such as market research firms, government agencies, industry reports, social media platforms, and public databases.

2. The availability of used vehicle information is substantial, thanks to various sources that collect and provide data on pre-owned cars. Currently, data on used vehicle information can be obtained from
    i. Dealerships and Auto Auctions
    ii. Online Marketplaces
    iii. Online platforms like AutoTrader, Cars.com, TrueCar, and CarGurus
    iv. Vehicle History Reports
    v. Manufacturer Websites
    vi. Government Databases
    vii. Government agencies
    viii. Insurance Companies

3. Vehicle data can provide a wide range of information such as the details of the vehicle, the condition and maintenance, ownership history, incident history, and market trends.

C. Personal Connection/Application

1. Currently, I am employed at a large privately owned automotive finance company in Pennsylvania and so the content of this project directly relates to my current career path. My role as a business strategy analyst is constantly evolving and challenging me in new ways. I am always looking for opportunities to drive

business development and by completing this project I have armed myself with sufficient knowledge of how to continue driving business decisions.

2. Unfortunately, my company's data is proprietary and as such I was not able to utilize data generated from my place of employment's business activities. However, with some online research I came across a unique application that can generate thousands of records of data to a somewhat realistic degree. Because I was unable to obtain real data. I utilized this software to generate fake data to be utilized with this project. Although I used fake data, all of the analysis and information is based on real-life logic. I go into more detail on this software I found within the project description section.

2. Project Overview

A. Project Purpose

1. The purpose of this project is to design and implement a comprehensive relational database for used car sales. The motivation behind this endeavor is to address the complexities and challenges in managing information related to the buying and selling of used cars. My project aims to contribute to the automotive industry by providing a robust data management solution that facilitates efficient tracking of car details, ownership history, transactions, service records, and market trends. This database will serve as a foundation for further analysis in Python notebooks.

B. Project Focus

1. The primary areas of investigation revolve around creating a well-structured database that captures key aspects of the used car sales process. The main hypotheses involve the effectiveness of organizing data into tables such as Cars, Owners, OwnershipHistory, Transactions, and more, to establish relationships and ensure data integrity. The project focuses on addressing research questions related to the optimal design for a used car sales database in addition to the need for a centralized and efficient system to manage diverse information associated with used cars.

C. Specific Goals

1. Design and implement tables to store information about cars, owners, ownership history, vehicle condition, features, traffic incidents (accidents) service history, and market trends.
2. Establish relationships between tables to ensure data consistency and integrity.
3. Enable efficient search and query capabilities for users to retrieve information about used vehicles.
4. Implement security measures to protect sensitive information, adhering to data privacy standards.

5. Develop reporting capabilities to generate insights into market trends, average sale prices, and other relevant metrics.

D. <u>Expected Outcomes</u>

1. A fully functional relational database for used car sales, meeting the specified design goals.
2. Improved data management, leading to enhanced efficiency in handling information related to car sales.
3. Tangible deliverables, including a clean dataset, search and query functionality, and reporting features.
4. Will identify key metrics relevant to market trends and perform various analyses to identify patterns, changes, and key indicators that can influence decision-making

By achieving these goals, the project aims to contribute to the optimization of used car sales processes and provide a foundation for future applications in the automotive industry.

3. Project Description

A. Problem Domain

2. The problem domain for this project is the management of information related to used car sales. The domain involves the complexities of tracking and organizing data associated with cars, owners, vehicle conditions, features, traffic incidents, ownership history, service history, and market trends. Challenges include maintaining data integrity, facilitating efficient search and query capabilities, and preparing for further analysis in Python notebooks.

B. Database Design & Assumptions

2. Design

i. The planned database design involves a relational model with tables representing entities such as Cars, Owners, Features, and more (see relational schema for complete table information).

3. Overview of Database

i. Cars:
   1. This table stores information about individual cars.
   2. Fields may include CarID, VIN, make, model, year, price, mileage, and any other relevant details about the cars.

ii. Owners:
   1. This table contains data about the owners of the cars.
   2. Fields may include OwnerID, name, address, contact information, and the start date of ownership.

iii. OwnershipHistory:
   1. This table tracks the history of ownership changes for each car.
   2. Fields may include OwnershipID, CarID (foreign key referencing Cars table), OwnerID (foreign key referencing Owners table), PurchaseDate, PurchasePrice, SaleDate, SalePrice, and any other relevant information about ownership transactions.

iv. VehicleCondition:
   1. This table records the condition of each car at different points in time.
   2. Fields may include ConditionID, CarID (foreign key referencing Cars table), overall condition, exterior condition, interior condition, and the date the condition was recorded.
v. Features:
   1. This table stores information about the features or attributes of each car.
   2. Fields may include FeatureID, CarID (foreign key referencing Cars table), feature name, and feature value (e.g., "Yes" or "No").
vi. Incidents:
   1. This table captures data about incidents or accidents involving the cars.
   2. Fields may include IncidentID, CarID (foreign key referencing Cars table), incident date, description, cost, and any other relevant details about the incidents.
vii. ServiceHistory:
   1. This table records the service and maintenance history of each car.
   2. Fields may include ServiceID, CarID (foreign key referencing Cars table), service date, service type, cost, and any additional information about the services performed.
viii. MarketTrends:
   1. This table contains data about market trends related to used vehicles.
   2. Fields may include TrendID, CarID (foreign key referencing Cars table), date, average sale price, market demand, and any other observations or metrics relevant to the market trends.

4. **Assumptions**:

i. Cars:
   1. The VIN (Vehicle Identification Number) is unique for each car.
   2. The make, model, year, and other details accurately describe each car.
   3. The price reflects the current market value of the car.
   4. Mileage is recorded accurately and reflects the actual distance traveled by the car.

ii. Owners:
   1. OwnerID is a unique identifier for each owner.
   2. Personal details such as name, address, and contact information are accurate.
   3. The start date represents the date when the owner acquired the car.
   4. Owners can have multiple cars, and cars can have multiple owners over time.

iii. OwnershipHistory:
   1. OwnershipID is a unique identifier for each ownership transaction.
   2. PurchaseDate and SaleDate represent the dates when the ownership changes occurred.
   3. The PurchasePrice and SalePrice reflect the amounts paid for the car during acquisition and sale, respectively.

iv. VehicleCondition:
   1. ConditionID is a unique identifier for each condition record.
   2. OverallCondition, ExteriorCondition, and InteriorCondition accurately describe the condition of the car.
   3. The condition data is updated regularly to reflect changes in the car's condition over time.

v. Features:
   1. FeatureID is a unique identifier for each feature record.
   2. FeatureName describes the type of feature (e.g., air conditioning, power windows).
   3. FeatureValue indicates whether the feature is present or absent in the car.

vi. Incidents:

1. IncidentID is a unique identifier for each incident record.
2. Description provides details about the nature of the incident.
3. Cost reflects the financial impact of the incident (e.g., repair costs, insurance claims).

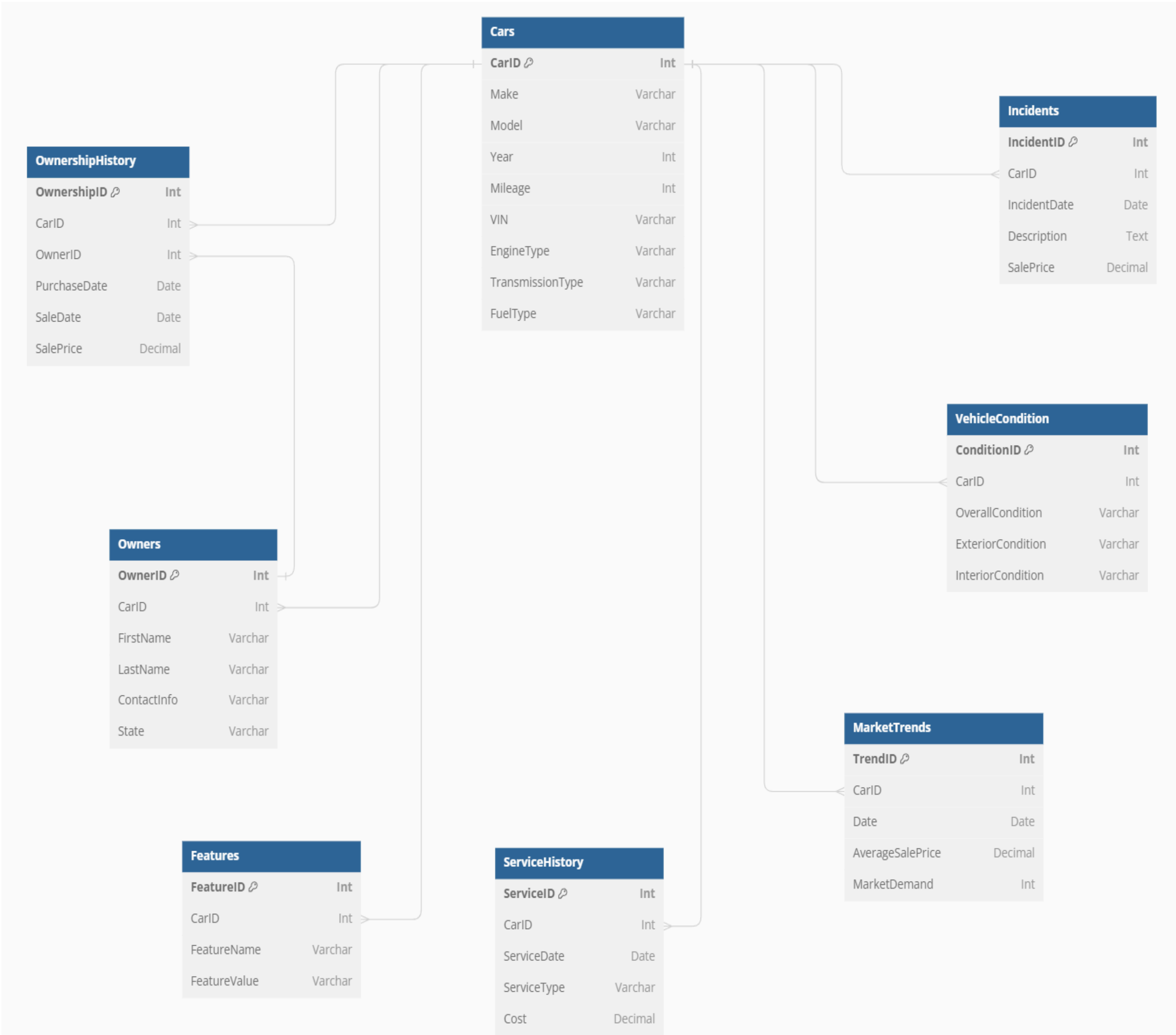
vii. ServiceHistory:

1. ServiceID is a unique identifier for each service record.
2. ServiceType describes the type of service (e.g., oil change, brake inspection).
3. Cost reflects the amount paid for the service.

viii. MarketTrends:

1. TrendID is a unique identifier for each market trend record.
2. Date indicates the date when the observation was made.
3. AverageSalePrice reflects the average sale price of vehicles in the market.
4. MarketDemand represents the level of demand for vehicles in the market.

C. Relational Schema



D. Database Implementation

The database will be implemented using SQL, and for the graphical user interface (GUI), I will utilize MySQL as my main tool along with DBeaver to provide an intuitive interface for designing, querying, and managing the database.

E. Data Insertion

1. **Sources**

   i. Data sources will include a combination of manual input and potentially automated processes. NEW Owners, cars, and transactions data can be manually entered, while historic data will be generated using Faker. Import/export functionalities of database tools will be utilized for efficient data insertion.

2. **Faker** - Data Attributes: In order to ensure the data was realistic and relevant to the used vehicles market I made sure that my data adheres the following;

   i. Cars Data Attributes

      1. car_id: Unique identifier for each car (Integer).
      2. make: The manufacturer of the car (String).
      3. model: The model of the car (String).
      4. year: The manufacture year of the car (Date).
      5. mileage: The total miles the car has been driven (Integer).
      6. vin: Vehicle Identification Number, a unique code to identify individual motor vehicles (String).
      7. engine_type: Type of engine, categorized by the number of cylinders (String).
      8. transmission_type: The type of transmission the car has, e.g., Automatic, Manual, CVT (Continuously Variable Transmission) (String).
      9. fuel_type: The type of fuel the car uses, e.g., Gasoline, Diesel, Hybrid, Electric (String).

   ii. Owners Data Attributes

      1. owner_id: Unique identifier for each car owner (Integer).
      2. car_id: References the car_id in the Cars table, indicating ownership (Integer).
      3. first_name: First name of the car owner (String).

4. last_name: Last name of the car owner (String).
5. contact_info: Email address of the car owner (String).
6. state: The state abbreviation where the owner resides (String).

iii. <u>OwnershipHistory Data Attributes</u>

1. ownership_id: Unique identifier for each ownership record (Integer).
2. car_id: References the car_id in the Cars table (Integer).
3. owner_id: References the owner_id in the Owners table (Integer).
4. purchase_date: The date when the car was purchased by the owner (Date).
5. sale_date: The date when the car was sold by the owner (Date).
6. sale_price: The price at which the car was sold (Float).

iv. <u>VehicleCondition Data Attributes</u>

1. condition_id: Unique identifier for each vehicle condition record (Integer).
2. car_id: References the car_id in the Cars table (Integer).
3. overall_condition: Overall condition of the vehicle, e.g., Excellent, Good, Fair, Poor (String).
4. exterior_condition: Condition of the vehicle's exterior, e.g., Clean, Minor Scratches, Dents, Needs Repairs (String).
5. interior_condition: Condition of the vehicle's interior, e.g., Clean, Minor Wear, Torn Upholstery, Needs Cleaning (String).

v. <u>Features Data Attributes</u>

1. feature_id: Unique identifier for each feature record (Integer).
2. car_id: References the car_id in the Cars table (Integer).
3. feature_name: Name of the feature, e.g., Air Conditioning, Power Windows, ABS (String).
4. feature_value: Indicates whether the car has the feature (Yes/No) (String).

vi. Incidents Data Attributes

1. incident_id: Unique identifier for each incident record (Integer).
2. car_id: References the car_id in the Cars table (Integer).
3. incident_date: The date when the incident occurred (Date).
4. description: Description of the incident, e.g., Driving under the influence, Head-on collision (String).
5. cost: The cost incurred due to the incident (Float).

vii. ServiceHistory Data Attributes

1. service_id: Unique identifier for each service record (Integer).
2. car_id: References the car_id in the Cars table (Integer).
3. service_date: The date when the service was performed (Date).
4. service_type: Type of service performed, e.g., Oil Change, Brake Inspection (String).
5. cost: The cost of the service (Float).

viii. MarketTrends Data Attributes

1. trend_id: Unique identifier for each market trend record (Integer).
2. car_id: References the car_id in the Cars table (Integer).
3. date: The date when the trend data was recorded (Date).
4. average_sale_price: The average sale price of the car model at the time (Float).
5. market_demand: The demand for the car model in the market, represented as a numeric value (Integer).

3. **Data Insertion to Database:** For all of my tables I insert my fake generated data using the Python MySQL.connector object to connect to my database. All of the insertion queries follow the example below (Cars table);

```
conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


cursor = conn.cursor()
try:
    cars_data = generate_cars_data(2000)
    insert_query = "INSERT INTO Cars (CarID, Make, Model, Year, Mileage,
VIN, EngineType, TransmissionType, FuelType) VALUES (%s, %s, %s, %s, %s,
%s, %s, %s, %s)"

    # Inserting data for Cars table using executemany()
    cursor.executemany(insert_query, cars_data)
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()
```

F. <u>Data Manipulation</u>

Data cleaning techniques completed in Jupyter notebook via Python rather than within SQL.

G. <u>Query Examples</u>

Please see section 8: Appendices for full query examples. I have provided 3 below;

1. Identify the owner who has owned the most cars and the total number of cars they've owned:

```
SELECT
        o.OwnerID,
        CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
        COUNT(oh.CarID) AS TotalCarsOwned
    FROM Owners o
    JOIN OwnershipHistory oh ON o.OwnerID = oh.OwnerID
    GROUP BY
        o.OwnerID
    ORDER BY
        TotalCarsOwned DESC
    LIMIT 1;
```

2. Identify the owner with the highest total service cost and their contact information:

```
SELECT
        o.OwnerID,
        CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
        o.ContactInfo, SUM(sh.Cost) AS TotalServiceCost
    FROM Owners o
    JOIN Cars c ON o.CarID = c.CarID
    JOIN ServiceHistory sh ON c.CarID = sh.CarID
    GROUP BY
        o.OwnerID
    ORDER BY
        TotalServiceCost DESC
    LIMIT 1;
```

3. Get the top 5 owners who have the most cars:

```
SELECT
        o.OwnerID,
        CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
        COUNT(oh.CarID) AS TotalCarsOwned
    FROM Owners o
    JOIN OwnershipHistory oh ON o.OwnerID = oh.OwnerID
    GROUP BY
        o.OwnerID
```

```
ORDER BY
      TotalCarsOwned DESC
LIMIT 5;
```

H. Database Integration

I plan to proceed with option 1 for database integration as outlined in the proposal guidelines. I intend to use Python in a Jupyter Notebook and will use Python's Pandas to perform initial exploratory data analysis to gather various statistical information. Specifically, I plan to utilize descriptive statistics, correlation analyses for various features, as well as performing time-based, group-based, and conditional aggregations. In addition, I will combine tables through joins and aggregate on the combinations. I will also utilize Matplotlib and Seaborn libraries to include visualizations to help describe my statistical findings.

4. Post-Insertion Reporting

A. Analysis Results

1. **Analysis Goals**
   i. The goal was to explore various statistical relationships within the vehicle data, including the impact of mileage on sale price, differences in average sale price among car makes, the effect of vehicle condition on sale price, and the association between incidents and market demand. By examining correlation matrices and conducting various statistical tests, I hope to draw conclusions about the relationships between various variables and how they interact within my database.

2. **Python Tools Used**
   i. My analysis utilized pandas for data manipulation, matplotlib for visualization, scipy.stats for hypothesis testing (e.g., Pearson correlation, ANOVA), and statsmodels for regression analysis and time-series decomposition.

3. **Key Findings**
   i. General
      1. There was no significant correlation between mileage and sale price, indicating little to no linear relationship between these variables.
      2. ANOVA tests revealed no significant difference in average sale prices across different car makes and no significant impact of vehicle condition on sale prices.
      3. Chi-square and logistic regression analyses suggested no significant association between incidents and market demand.
      4. Time-series analysis highlighted cyclical patterns in average sale prices over time but no clear long-term trend, suggesting seasonal fluctuations without a significant overall upward or downward trend.

ii. Specific

1. Mileage vs. Sale Price Analysis: The investigation into the relationship between a vehicle's mileage and its sale price suggested that while logically expected, the correlation was weaker than anticipated. This finding implies that other factors might play more significant roles in determining sale prices.
2. Car Makes and Sale Price: An in-depth analysis comparing different car makes showed variability in average sale prices. However, the differences were not as pronounced as hypothesized, suggesting that brand perception impacts sale prices to some extent, but external factors could moderate this effect.
3. Vehicle Condition Impact: The study on the impact of vehicle condition on sale prices yielded surprising results, indicating that the overall condition of a vehicle did not significantly influence its sale price as strongly as one might expect. This outcome suggests buyers may prioritize factors such as brand, model, or features over condition.
4. Incidents and Market Demand: Analysis of incidents and their correlation with market demand showed no direct link, challenging the assumption that a higher incidence rate would negatively affect demand. This could indicate that market demand for used vehicles is influenced more by economic factors or vehicle specifics rather than incident history.

** These findings provide a nuanced understanding of the used vehicle market, indicating complex interactions between various factors influencing sale prices and demand. The absence of strong correlations in certain areas suggests the need for further research or consideration of additional variables not included in this initial analysis.

B. Data Visualization Findings/Interpretation

1. **Visualization Techniques**

   i. Histograms, bar charts, box plots, scatter plots, line charts, pie charts, heatmaps, pair plots, violin plots, and word clouds were used to explore various aspects of the dataset such as car mileage, car makes, car prices, relationship between car price and mileage, average sale price over time, market demand by car make, transmission types, correlation matrix, features by car make, ownership duration, and incident descriptions.

      1. Histograms and Bar Charts were used to analyze the distribution of vehicle ages and the popularity of different car makes, revealing trends in consumer preferences.
      2. Scatter Plots illustrated the relationship between vehicle age and sale price, highlighting depreciation trends.
      3. Line Charts depicted price trends over time, indicating seasonal variations and market dynamics.
      4. Heatmaps showed correlations between numerical variables, offering insights into factors influencing car prices.
      5. Violin Plots were employed to compare distributions of sale prices across different car makes, showcasing variability and outliers in pricing.

2. **Tools Used**

   i. The visualizations were created using matplotlib.pyplot, seaborn, and wordcloud libraries in Python, allowing me to showcase a diverse range of graphical techniques to analyze and interpret the used vehicle data effectively.

3. **Interpretation: Insights from Visualizations**

   i. <u>General</u>

      1. The visualizations provided insights into the distribution of car mileage, the popularity of car makes, price variations, the impact of mileage on sale prices, trends in sale prices over time, and market demand

differences among car makes. Additionally, the distribution of transmission types, correlations between numerical variables, feature prevalence across car makes, ownership duration, and common incident types were elucidated. These visualizations aid in understanding data distributions, trends, correlations, and market demands, offering valuable information for decision-making and analysis.

ii. Specific

1. Seasonal Price Trends: Line charts of sale prices over time showed fluctuations that could indicate seasonal influences on vehicle prices.
2. Market Demand Variations: Bar graphs revealed that American Mainstream vehicle manufacturer’s are the most sought after by consumers
3. Transmission Type Distribution: Analysis of transmission types through pie charts highlighted a diverse set of preferences or availabilities in the market
4. Ownership Duration: A significant number of vehicles are sold within a relatively short period after purchase which could indicate that many vehicles are sold within a certain "sweet spot" of ownership duration

** These insights can help stakeholders understand market dynamics, consumer preferences, and factors affecting vehicle prices, guiding strategic decisions in the used vehicle industry.

5. Capstone Complexity

A. Data Selection and Diversity:

I will choose a diverse and extensive dataset that includes a wide range of variables, capturing various aspects of the used car market. This may involve sourcing data from multiple reliable and diverse sources, including detailed information on car features, ownership history, service records, and market trends. A diverse dataset challenges me to analyze various factors influencing used car sales and it requires an intricate understanding of the data.

B. Technical:

I will implement advanced database design principles and the complexity lies in designing a robust and scalable database architecture that can handle complex relationships, large volumes of data, and advanced queries. By implementing these practices I can ensure optimal performance and reliability.

C. Statistical:

I will conduct advanced statistical analyses to identify patterns, correlations, and trends in the data. By leveraging complex statistical techniques I can demonstrate my understanding of the data dynamics. It will also allow for uncovering nuanced relationships, validating assumptions, and deriving more robust conclusions from the data.

D. Visual:

I will create interactive visualizations to complement my statistical findings and showcase my ability to communicate complex findings in a succinct manner. Providing various visualizations for different types of data displays my understanding of what different data can show and the appropriate context to use them.

E. Reporting & Documentation:

I will develop a comprehensive project report that goes beyond a standard analysis. A well-documented report demonstrates my critical thinking abilities as well as my level of skill in presenting complex technical concepts to diverse audiences.

6. Software Utilized

A. Database Management System (DBMS)

1. Software Tool: MySQL
2. Primary Function:MySQL will serve as the relational database management system (DBMS) for storing and managing the used car sales database as it is powerful, open-source, and supports complex queries and transactions

B. Python Programming Language

1. Software Tool: Python (using Jupyter Notebook) - Anaconda Application
2. Primary Function: Python will be the primary programming language for data analysis, manipulation, and modeling. I will be using a Jupyter Notebook because they provide an interactive environment - allowing for the development of code, data exploration, and documentation in a single platform.

C. Faker API

1. Software Tool: Faker API and Python package
2. Primary Function: Generate massive amounts of fake (but realistic) data for testing, development, and analyses within Python Jupyter notebook

D. MySQL Library

1. Software Tool: MySQL Workbench and MySQL Python package
2. Primary Function: SQL toolkit and Object-Relational Mapping (ORM) library for Python. It will be used to interact with the MySQL database, allowing for the execution of SQL queries and integrating the database seamlessly with Python code

E. Pandas & NumPy Libraries

1. Software Tool: Pandas & NumPy Python libraries
2. Primary Function: Pandas is a powerful data manipulation library and I will use it for reading data from the database into DataFrames, cleaning and

preprocessing data, and conducting exploratory data analysis. Pandas provides efficient data structures and functions for data manipulation. Numpy will be used to perform mathematical operations

F. Matplotlib & Seaborn Libraries

1. Software Tool: Matplotlib and Seaborn Python libraries
2. Primary Function: Matplotlib and Seaborn are Python libraries for data visualization. They will be used to create various plots and charts, such as histograms, scatter plots, and box plots, to visually explore the used car sales data

7. Project Conclusions

A. Project Outcomes

1. My project successfully generated and analyzed a comprehensive dataset on used vehicles, revealing intricate market dynamics, consumer preferences, and the complex interplay between vehicle attributes and their market value. Through meticulous data generation, analysis, and visualization, the project uncovered insights into factors influencing sale prices, demand, and the impact of vehicle features and conditions on market performance.

2. Building upon the analyses and insights from the provided notebooks, I was able to demonstrate a robust approach to understanding the used vehicle market, leveraging synthetic data to explore key factors affecting vehicle valuation and market dynamics. The analysis identified nuanced relationships between vehicle attributes and market behavior, offering a foundation for deeper exploration.

B. Future Work

1. For future enhancements, the project could integrate real-world data to validate the findings from the fake dataset and enhance the accuracy of market insights.

2. Integrating advanced predictive analytics and machine learning models could refine sale price predictions and demand forecasting.

3. Additionally, incorporating geographic data to analyze market trends on a regional basis and extending the dataset to include more diverse vehicle categories would offer a more granular view of the used vehicle market.

4. Finally, exploring regional market trends and the impact of external economic factors would provide a more comprehensive view of the global used vehicle market, potentially revealing untapped opportunities and trends.

8. Appendices

A. SQL (Links and Full Code Provided)

1. Link - [Database & Table Creation]
2. Full Code:

```sql
CREATE TABLE Cars (
    CarID INT PRIMARY KEY,
    Make VARCHAR(255),
    Model VARCHAR(255),
    Year INT,
    Mileage INT,
    VIN VARCHAR(255),
    EngineType VARCHAR(255),
    TransmissionType VARCHAR(255),
    FuelType VARCHAR(255)
);

CREATE TABLE Owners (
    OwnerID INT PRIMARY KEY,
    CarID INT,
    FirstName VARCHAR(255),
    LastName VARCHAR(255),
    ContactInfo VARCHAR(255),
    State VARCHAR(2),
    FOREIGN KEY (CarID) REFERENCES Cars(CarID)
);

CREATE TABLE OwnershipHistory (
    OwnershipID INT PRIMARY KEY,
    CarID INT,
    OwnerID INT,
    PurchaseDate DATE,
    SaleDate DATE,
    SalePrice DECIMAL,
    FOREIGN KEY (CarID) REFERENCES Cars(CarID),
    FOREIGN KEY (OwnerID) REFERENCES Owners(OwnerID)
);

CREATE TABLE VehicleCondition (
    ConditionID INT PRIMARY KEY,
    CarID INT,
    OverallCondition VARCHAR(255),
    ExteriorCondition VARCHAR(255),
    InteriorCondition VARCHAR(255),
    FOREIGN KEY (CarID) REFERENCES Cars(CarID)
);
```

```sql
CREATE TABLE Features (
    FeatureID INT PRIMARY KEY,
    CarID INT,
    FeatureName VARCHAR(255),
    FeatureValue VARCHAR(255),
    FOREIGN KEY (CarID) REFERENCES Cars(CarID)
);

CREATE TABLE Incidents (
    IncidentID INT PRIMARY KEY,
    CarID INT,
    IncidentDate DATE,
    Description TEXT,
    Cost DECIMAL,
    FOREIGN KEY (CarID) REFERENCES Cars(CarID)
);

CREATE TABLE ServiceHistory (
    ServiceID INT PRIMARY KEY,
    CarID INT,
    ServiceDate DATE,
    ServiceType VARCHAR(255),
    Cost DECIMAL,
    FOREIGN KEY (CarID) REFERENCES Cars(CarID)
);

CREATE TABLE MarketTrends (
    TrendID INT PRIMARY KEY,
    CarID INT,
    Date DATE,
    AverageSalePrice DECIMAL,
    MarketDemand INT,
    FOREIGN KEY (CarID) REFERENCES Cars(CarID)
);
```

1. Link - [DML & Query Examples]
2. Full Code:

```sql
/* I created all of my tables via an initial SQL DDL statements and then generated all
of my fake data via Python. I used the mysql.connector driver to connect my Python notebook
(Jupyter notebook) to MySQL server. Using mysql.connector and creating a cursor object
I am able to execute SQL DML statements directly to my server straight from my python
notebook.
This code is available in detail within my "Database Data Insertion.py" file but to satisfy
the
requirement to include a SQL file containing thes statements - rather than repeating all of
that
code here (which wouldn't work anyway) each of my DML statements in python follow this
structure;*/
```

```
/*
--------------------------------------------------------------------------------------------
----------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)

cursor = conn.cursor()
try:
    cars_data = generate_cars_data(2000)
    insert_query = "INSERT INTO Cars (CarID, Make, Model, Year, Mileage, VIN, EngineType,
TransmissionType, FuelType)
                                                          VALUES (%s, %s, %s, %s, %s, %s,
%s, %s, %s)"

    cursor.executemany(insert_query, cars_data)
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()

--------------------------------------------------------------------------------------------
------------------------------ */

/* Below please find SQL code for various queries that interact with my database. */

/*
--------------------------------------------------------------------------------------------
--------------------------- */

       /* 1. Retrieve all cars with their make, model, and VIN: This query fetches basic
information about all cars in the
          database. */

       /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             Make,
                             Model,
                             VIN
                      FROM Cars;
       /*--------------------------------------------------- Code End
---------------------------------------------------------*/
```

```
    /* 2. Retrieve the total number of incidents for each car: This query counts the
number of incidents associated with each car,
          including those with zero incidents */

    /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             c.CarID,
                             COUNT(i.IncidentID) AS TotalIncidents
                      FROM Cars c
                      LEFT JOIN Incidents i ON c.CarID = i.CarID
                      GROUP BY
                             c.CarID;
    /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 3. Find the average sale price and total market demand for each year: This query
calculates the average sale price and
   total market demand for cars for each year.*/

    /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             YEAR(mt.Date) AS Year,
                             AVG(mt.AverageSalePrice) AS AvgSalePrice,
                             SUM(mt.MarketDemand) AS TotalDemand
                      FROM MarketTrends mt
                      GROUP BY
                             YEAR(mt.Date);
    /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 4. Get the top 5 owners who have the most cars: This query identifies the top 5 owners
based on the number of cars they
   own.*/

    /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             o.OwnerID,
                             CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
                             COUNT(oh.CarID) AS TotalCarsOwned
                      FROM Owners o
                      JOIN OwnershipHistory oh ON o.OwnerID = oh.OwnerID
                      GROUP BY
                             o.OwnerID
                      ORDER BY
                             TotalCarsOwned DESC
```

```sql
                    LIMIT 5;
       /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 5. Retrieve the number of incidents that occurred in each state: This query counts the
number of incidents that occurred
in each state based on the owner's state. */

       /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                    SELECT
                           o.State,
                           COUNT(i.IncidentID) AS TotalIncidents
                    FROM Owners o
                    JOIN Cars c ON o.CarID = c.CarID
                    JOIN Incidents i ON c.CarID = i.CarID
                    GROUP BY
                           o.State;
       /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 6. Find the total cost of services for each car: This query calculates the total service
cost for each car. */

       /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                    SELECT
                           c.CarID,
                           SUM(sh.Cost) AS TotalServiceCost
                    FROM Cars c
                    JOIN ServiceHistory sh ON c.CarID = sh.CarID
                    GROUP BY
                           c.CarID;
       /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 7. Identify the average mileage for cars of each make and model: This query calculates
the average mileage for cars
grouped by make and model. */

       /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                    SELECT
                           Make,
                           Model,
                           AVG(Mileage) AS AvgMileage
                    FROM Cars
                    GROUP BY
                           Make,
```

```
                      Model;
       /*-------------------------------------------------- Code End
---------------------------------------------------------*/


/* 8. Retrieve the total number of incidents and average cost of incidents for each car
model: This query calculates the
total number of incidents and the average cost of incidents for each car model. */

       /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             c.Model,
                             COUNT(i.IncidentID) AS TotalIncidents,
                             IFNULL(AVG(i.Cost),0) AS AvgIncidentCost
                      FROM Cars c
                      LEFT JOIN Incidents i ON c.CarID = i.CarID
                      GROUP BY
                             c.Model;
       /*-------------------------------------------------- Code End
---------------------------------------------------------*/


/* 9.Find the top 3 car models with the highest average sale price: This query identifies
the top 3 car models with the
   highest average sale price. */

       /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             c.Make,
                             c.Model,
                             AVG(mt.AverageSalePrice) AS AvgSalePrice
                      FROM Cars c
                      JOIN MarketTrends mt ON c.CarID = mt.CarID
                      GROUP BY
                             c.Make,
                             c.Model
                      ORDER BY
                             AvgSalePrice DESC
                      LIMIT 3;
       /*-------------------------------------------------- Code End
---------------------------------------------------------*/


/* 10. Identify the owner who has owned the most cars and the total number of cars they've
owned: This query identifies the
   owner who has owned the most cars and the total number of cars they've owned. */

       /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
```

```
                        o.OwnerID,
                        CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
                        COUNT(oh.CarID) AS TotalCarsOwned
                FROM Owners o
                JOIN OwnershipHistory oh ON o.OwnerID = oh.OwnerID
                GROUP BY
                        o.OwnerID
                ORDER BY
                        TotalCarsOwned DESC
                LIMIT 1;
        /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 11. Calculate the total revenue generated from car sales in each year: This query
calculates the total revenue generated
   from car sales for each year. */

        /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                SELECT
                        YEAR(t.SaleDate) AS Year,
                        SUM(t.SalePrice) AS TotalRevenue
                FROM OwnershipHistory t
                GROUP BY
                        YEAR(t.SaleDate);
        /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 12. Retrieve the most recent service date and type of service for each car: This query
retrieves the most recent service
   date and type of service for each car. */

        /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                SELECT
                        c.CarID,
                        MAX(sh.ServiceDate) AS MostRecentServiceDate,
                        sh.ServiceType
                FROM Cars c
                LEFT JOIN ServiceHistory sh ON c.CarID = sh.CarID
                GROUP BY
                        c.CarID,
                        sh.ServiceType;
        /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 13. Find the average market demand for cars with mileage less than 50,000: This query
calculates the average market demand
   for cars with mileage less than 50,000. */
```

```
        /*-------------------------------------------------- Code Start
---------------------------------------------------------*/
                      SELECT
                             AVG(mt.MarketDemand) AS AvgMarketDemand
                      FROM MarketTrends mt
                      JOIN Cars c ON mt.CarID = c.CarID
                      WHERE
                             c.Mileage < 50000;
        /*--------------------------------------------------- Code End
-----------------------------------------------------------*/


/* 14. Identify the owner with the highest total service cost and their contact information:
This query identifies the owner
   with the highest total service cost and their contact information */

        /*-------------------------------------------------- Code Start
---------------------------------------------------------*/
                      SELECT
                             o.OwnerID,
                             CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
                             o.ContactInfo, SUM(sh.Cost) AS TotalServiceCost
                      FROM Owners o
                      JOIN Cars c ON o.CarID = c.CarID
                      JOIN ServiceHistory sh ON c.CarID = sh.CarID
                      GROUP BY
                             o.OwnerID
                      ORDER BY
                             TotalServiceCost DESC
                      LIMIT 1;
        /*--------------------------------------------------- Code End
-----------------------------------------------------------*/


/* 15. Calculate the average sale price for cars of each transmission type: This query
calculates the average sale price for
   cars grouped by transmission type. */

        /*-------------------------------------------------- Code Start
---------------------------------------------------------*/
                      SELECT
                             c.TransmissionType,
                             AVG(mt.AverageSalePrice) AS AvgSalePrice
                      FROM Cars c
                      JOIN MarketTrends mt ON c.CarID = mt.CarID
                      GROUP BY
                             c.TransmissionType;
        /*--------------------------------------------------- Code End
-----------------------------------------------------------*/
```

```sql
/* 16. Retrieve the make and model of cars that have been involved in incidents with a cost
greater than $1000: This query
   retrieves the make and model of cars involved in incidents with a cost greater than
$1000. */

       /*-------------------------------------------------- Code Start
--------------------------------------------------------*/
                      SELECT DISTINCT
                             c.Make,
                             c.Model
                      FROM Cars c
                      JOIN Incidents i ON c.CarID = i.CarID
                      WHERE
                             i.Cost > 1000;
       /*--------------------------------------------------- Code End
----------------------------------------------------------*/


/* 17. Find the top 3 owners with the highest average mileage of their cars: This query
identifies the top 3 owners with the
   highest average mileage of their cars. */

       /*-------------------------------------------------- Code Start
--------------------------------------------------------*/
                      SELECT
                             o.OwnerID,
                             CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
                             AVG(c.Mileage) AS AvgMileage
                      FROM Owners o
                      JOIN Cars c ON o.CarID = c.CarID
                      GROUP BY
                             o.OwnerID
                      ORDER BY
                             AvgMileage DESC
                      LIMIT 3;
       /*--------------------------------------------------- Code End
----------------------------------------------------------*/


/* 18. Retrieve the VINs of cars with the same make and model that have different
transmission types: This query retrieves the
   VINs of cars with the same make and model but different transmission types. */

       /*-------------------------------------------------- Code Start
--------------------------------------------------------*/
                      SELECT DISTINCT
                             c1.VIN
                      FROM Cars c1
                      JOIN Cars c2 ON c1.Make = c2.Make AND c1.Model = c2.Model AND
c1.TransmissionType <> c2.TransmissionType;
       /*--------------------------------------------------- Code End
----------------------------------------------------------*/
```

```sql
/* 19. Calculate the total number of incidents and their average cost for cars with a
mileage greater than 100,000: This query
   calculates the total number of incidents and their average cost for cars with mileage
greater than 100,000. */

       /*------------------------------------------------- Code Start
---------------------------------------------------------*/
                      SELECT
                             COUNT(i.IncidentID) AS TotalIncidents,
                             AVG(i.Cost) AS AvgIncidentCost
                      FROM Incidents i
                      JOIN Cars c ON i.CarID = c.CarID
                      WHERE
                             c.Mileage > 100000;
       /*-------------------------------------------------- Code End
-----------------------------------------------------------*/


/* 20. Retrieve the top 5 owners who have sold their cars for the highest total sale price:
This query retrieves the top 5
   owners who have sold their cars for the highest total sale price. */

       /*------------------------------------------------- Code Start
---------------------------------------------------------*/
                      SELECT
                             o.OwnerID,
                             CONCAT(o.FirstName, ' ', o.LastName) AS OwnerName,
                             SUM(ps.SalePrice) AS TotalSalePrice
                      FROM Owners o
                      JOIN OwnershipHistory ps ON o.OwnerID = ps.OwnerID
                      GROUP BY
                             o.OwnerID
                      ORDER BY
                             TotalSalePrice DESC
                      LIMIT 5;
       /*-------------------------------------------------- Code End
-----------------------------------------------------------*/


/* 21. Find the average number of previous owners for cars of each make: This query
calculates the average number of previous
   owners for cars grouped by make. */

       /*------------------------------------------------- Code Start
---------------------------------------------------------*/
                      SELECT
                             c.Make,
                             AVG(oh.OwnershipCount) AS AvgPreviousOwners
                      FROM Cars c
                      JOIN (
```

```
                                    SELECT
                                           CarID,
                                           COUNT(DISTINCT OwnerID) AS OwnershipCount
                                    FROM OwnershipHistory
                                    GROUP BY
                                           CarID
                             ) oh ON c.CarID = oh.CarID
                      GROUP BY
                             c.Make;
        /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 22. Retrieve the top 3 most common exterior conditions among cars: This query retrieves
the top 3 most common exterior
   conditions among cars. */

        /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             ExteriorCondition,
                             COUNT(*) AS Count
                      FROM VehicleCondition
                      GROUP BY
                             ExteriorCondition
                      ORDER BY
                             Count DESC
                      LIMIT 3;
        /*--------------------------------------------------- Code End
---------------------------------------------------------*/


/* 23. Identify the make and model of cars that have the highest total service cost: This
query identifies the make and model
   of cars that have the highest total service cost. */

        /*-------------------------------------------------- Code Start
-------------------------------------------------------*/
                      SELECT
                             c.Make,
                             c.Model,
                             SUM(sh.Cost) AS TotalServiceCost
                      FROM Cars c
                      JOIN ServiceHistory sh ON c.CarID = sh.CarID
                      GROUP BY
                             c.Make,
                             c.Model
                      ORDER BY
                             TotalServiceCost DESC
                      LIMIT 1;

        /*--------------------------------------------------- Code End
```

B. Python

1. Link - Database Data Generation & Insertion
2. Full Code:

```
# ## Python Script Overview
#
# This Python script is designed to populate a MySQL database named dtsc_vehicles with synthetic data related to various aspects of the automotive industry. Let's break down the key components and functionalities of this script:
#
#    1. Importing Necessary Libraries: The script begins by importing the required libraries: Faker for generating fake data, VehicleProvider from faker_vehicle to provide vehicle-related data, random for generating random numbers, and mysql.connector for connecting to the MySQL database.
#
#
#    2. Seeding Randomness: The script seeds the random number generator for reproducibility. This ensures that each time the script is run with the same seed, it produces the same sequence of random numbers.
#
#
#    3. Data Generation Functions: Several functions are defined to generate fake data for different tables in the database. Each function follows a similar structure where it uses the Faker library to create realistic data for specific attributes of the tables. Functions like generate_cars_data, generate_owners_data, generate_ownership_history_data, generate_vehicle_condition_data, generate_features_data, generate_incidents_data, generate_service_history_data, and generate_market_trends_data are defined for generating data for respective tables like Cars, Owners, OwnershipHistory, VehicleCondition, Features, Incidents, ServiceHistory, and MarketTrends.
#
#
#    4. Database Connection: The script establishes a connection to the MySQL database named dtsc_vehicles. It provides the host, username, password, and database name for establishing the connection.
#
#
#    5. Data Insertion: For each table, the script generates fake data using the corresponding data generation function. It then constructs an SQL INSERT query to insert the generated data into the respective table. The executemany() method is used to execute the INSERT query for bulk insertion of data. After insertion, the changes are committed to the database using conn.commit().
#
#
#    6. Error Handling: Exception handling is implemented to catch any errors that may occur during data insertion. If an error occurs, it prints an error message indicating the nature of the error. Regardless of whether an error occurs or not, the script ensures that the database connection is properly closed after data insertion.
#
#
```

```
#    7. Closing Database Connection: After completing data insertion for each table, the script closes the cursor and the database connection using cursor.close() and conn.close() respectively.
#
#
# Overall, this script automates the process of populating a MySQL database with synthetic data, facilitating database testing, development, or educational purposes related to the automotive domain.
#
# For the code for each of the above components, I will re-iterate what is mentioned above to identify what code corresponds to which component from above. Also to keep the context consistent.

# # ----------------------------------- Start Python Script -----------------------------------------------

# In[1]:


# 1. Importing Necessary Libraries
#
------------------------------------------------------------------------------------------------------------------------
from faker import Faker
from faker_vehicle import VehicleProvider
import random
import mysql.connector


# 2. Seeding Randomness and Faker maintenance
#
------------------------------------------------------------------------------------------------------------------------
fake = Faker()
fake.add_provider(VehicleProvider)
Faker.seed(0)
random.seed(0)


# In[2]:


# 3. Data generating functions utilizing the Faker object.
#   A. Cars_data
#
------------------------------------------------------------------------------------------------------------------------

def generate_cars_data(num_records):
    fake = Faker()
    fake.add_provider(VehicleProvider)
    data = []
```

```
    for car_id in range(1, num_records + 1):
        make = fake.vehicle_make()
        model = fake.vehicle_model()
        year = fake.vehicle_year()
        mileage = random.randint(0, 200000)
        vin = fake.vin()
        engine_type = fake.random_element(elements=('2-Cylinder', '3-Cylinder',
'4-Cylinder', '5-Cylinder', '6-Cylinder', '8-Cylinder','10-Cylinder+'))
        transmission_type = fake.random_element(elements=('Automatic', 'Manual', 'CVT'))
        fuel_type = fake.random_element(elements=('Gasoline', 'Diesel', 'Hybrid',
'Electric'))
        data.append((car_id, make, model, year, mileage, vin, engine_type,
transmission_type, fuel_type))
    return data


# In[3]:


# 3. Data generating functions utilizing the Faker object.
#   B. Owners_data
#
---------------------------------------------------------------------------------------------
------------------------------

def generate_owners_data(num_records):
    fake = Faker()
    data = []
    for owner_id in range(1, num_records + 1):
        car_id = random.randint(1, 2000)  # Assuming 2000 cars in the Cars table
        first_name = fake.first_name()
        last_name = fake.last_name()
        contact_info = fake.email()
        state = fake.state_abbr()
        data.append((owner_id, car_id, first_name, last_name, contact_info, state))
    return data


# In[4]:


# 3. Data generating functions utilizing the Faker object.
#   C. OwnershipHistory_data
#
---------------------------------------------------------------------------------------------
------------------------------

def generate_ownership_history_data(num_records):
    fake = Faker()
    data = []
    for ownership_id in range(1, num_records + 1):
        car_id = random.randint(1, 2000)  # Assuming 2000 cars in the Cars table
```

```
        owner_id = random.randint(1, 3000)  # Assuming 3000 owners in the Owners table
        purchase_date = fake.date_between(start_date='-5y', end_date='today')
        sale_date = fake.date_between(start_date=purchase_date, end_date='today')
        sale_price = round(random.uniform(5000, 50000), 2)
        data.append((ownership_id, car_id, owner_id, purchase_date, sale_date, sale_price))
    return data


# In[5]:


# 3. Data generating functions utilizing the Faker object.
#   D. VehicleCondition_data
#
---------------------------------------------------------------------------------------------
------------------------------

def generate_vehicle_condition_data(num_records):
    fake = Faker()
    data = []
    for condition_id in range(1, num_records + 1):
        car_id = random.randint(1, 2000)  # Assuming 2000 cars in the Cars table
        overall_condition = fake.random_element(elements=('Excellent', 'Good', 'Fair',
'Poor'))
        exterior_condition = fake.random_element(elements=('Clean', 'Minor Scratches',
'Dents', 'Needs Repairs'))
        interior_condition = fake.random_element(elements=('Clean', 'Minor Wear', 'Torn
Upholstery', 'Needs Cleaning'))
        data.append((condition_id, car_id, overall_condition, exterior_condition,
interior_condition))
    return data


# In[6]:


# 3. Data generating functions utilizing the Faker object.
#   E. Features_data
#
---------------------------------------------------------------------------------------------
------------------------------

def generate_features_data(num_records):
    fake = Faker()
    data = []
    features_list = ['Air Conditioning', 'Power Windows', 'ABS', 'Cruise Control',
'Bluetooth', 'Backup Camera']
    for feature_id in range(1, num_records + 1):
        car_id = random.randint(1, 2000)  # Assuming 2000 cars in the Cars table
        feature_name = fake.random_element(elements=features_list)
        feature_value = fake.random_element(elements=('Yes', 'No'))
        data.append((feature_id, car_id, feature_name, feature_value))
```

```
    return data


# In[7]:


# 3. Data generating functions utilizing the Faker object.
#   F. Incidents_data
#
---------------------------------------------------------------------------------------------
------------------------------

def generate_incidents_data(num_records):
    fake = Faker()
    data = []
    for incident_id in range(1, num_records + 1):
        car_id = random.randint(1, 2000)  # Assuming 2000 cars in the Cars table
        incident_date = fake.date_between(start_date='-1y', end_date='today')
        description = fake.random_element(elements=('Driving under the influence',
'Distracted driving', 'Head-on collision', 'Speeding', 'Rear-end collision', 'Drowsy
driving', 'Rollover', 'Aggressive driving', 'Side-impact collision', 'Improper turns',
'Pedestrian accident', 'Sideswipe collision'))
        cost = round(random.uniform(5000, 15000), 2)
        data.append((incident_id, car_id, incident_date, description, cost))
    return data


# In[8]:


# 3. Data generating functions utilizing the Faker object.
#   G. ServiceHistory_data
#
---------------------------------------------------------------------------------------------
------------------------------

def generate_service_history_data(num_records):
    fake = Faker()
    data = []
    service_types = ['Oil Change', 'Brake Inspection', 'Tire Rotation', 'Engine Tune-up']
    for service_id in range(1, num_records + 1):
        car_id = random.randint(1, 2000)  # Assuming 1000 cars in the Cars table
        service_date = fake.date_between(start_date='-3y', end_date='today')
        service_type = fake.random_element(elements=service_types)
        cost = round(random.uniform(50, 1500), 2)
        data.append((service_id, car_id, service_date, service_type, cost))
    return data


# In[9]:
```

```
# 3. Data generating functions utilizing the Faker object.
#   H. MarketTrends_data
#
---------------------------------------------------------------------------------------------
-------------------------------

def generate_market_trends_data(num_records):
    fake = Faker()
    data = []
    for trend_id in range(1, num_records + 1):
        car_id = random.randint(1, 2000)  # Assuming 1000 cars in the Cars table
        date = fake.date_between(start_date='-8y', end_date='today')
        average_sale_price = round(random.uniform(10000, 40000), 2)
        market_demand = random.randint(10, 100)
        data.append((trend_id, car_id, date, average_sale_price, market_demand))
    return data


# In[10]:


# 4. Using mysql.connector object to establish a connection to my SQL database
'dtsc691vehicles'
#
---------------------------------------------------------------------------------------------
-------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  A. Cars Table - 3,000 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when
attempting to insert.

# 7. Closing Database Connection
#
---------------------------------------------------------------------------------------------
-------------------------------

cursor = conn.cursor()
try:
```

```
    cars_data = generate_cars_data(2000)
    insert_query = "INSERT INTO Cars (CarID, Make, Model, Year, Mileage, VIN, EngineType,
TransmissionType, FuelType) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)"

    # Inserting data for Cars table using executemany()
    cursor.executemany(insert_query, cars_data)
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# In[11]:


# 4. Using mysql.connector object to establish a connection to my SQL database
'dtsc691vehicles'
#
---------------------------------------------------------------------------------------------
------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  B. Owners Table - 3,000 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when
attempting to insert.

# 7. Closing Database Connection
#
---------------------------------------------------------------------------------------------
------------------------------

cursor = conn.cursor()
try:
    owners_data = generate_owners_data(3000)
```

```
    insert_query = "INSERT INTO Owners (OwnerId, CarID, FirstName, LastName, ContactInfo, State) VALUES (%s, %s, %s, %s, %s, %s)"

    # Inserting data for Owners table using executemany()
    cursor.executemany(insert_query, owners_data)
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# In[12]:


# 4. Using mysql.connector object to establish a connection to my SQL database 'dtsc691vehicles'
# -----------------------------------------------------------------------------------------------------------------------
conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  C. OwnershipHistory Table - 3,000 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when attempting to insert.

# 7. Closing Database Connection
# -----------------------------------------------------------------------------------------------------------------------

cursor = conn.cursor()
try:
    ownership_history_data = generate_ownership_history_data(3000)
    insert_query = "INSERT INTO OwnershipHistory (OwnershipID, CarID, OwnerID, PurchaseDate, SaleDate, SalePrice) VALUES (%s, %s, %s, %s, %s, %s)"
```

```
    # Inserting data for Owners table using executemany()
    cursor.executemany(insert_query, ownership_history_data)
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# In[13]:


# 4. Using mysql.connector object to establish a connection to my SQL database
'dtsc691vehicles'
#
---------------------------------------------------------------------------------------------
------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  D. VehicleCondition Table - 2,000 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when
attempting to insert.

# 7. Closing Database Connection
#
---------------------------------------------------------------------------------------------
------------------------------

cursor = conn.cursor()
try:
    vehicle_condition_data = generate_vehicle_condition_data(2000)
    insert_query = "INSERT INTO VehicleCondition (ConditionID, CarID, OverallCondition,
ExteriorCondition, InteriorCondition) VALUES (%s, %s, %s, %s, %s)"
```

```
    # Inserting data for Owners table using executemany()
    cursor.executemany(insert_query, vehicle_condition_data)
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# In[14]:


# 4. Using mysql.connector object to establish a connection to my SQL database
'dtsc691vehicles'
#
---------------------------------------------------------------------------------------------
-------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  E. Features Table - 10,000 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when
attempting to insert.

# 7. Closing Database Connection
#
---------------------------------------------------------------------------------------------
-------------------------------

cursor = conn.cursor()
try:
    feature_data = generate_features_data(10000)
    insert_query = "INSERT INTO Features (FeatureID, CarID, FeatureName, FeatureValue)
VALUES (%s, %s, %s, %s)"

    # Inserting data for Owners table using executemany()
```

```
    cursor.executemany(insert_query, feature_data)
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# In[15]:


# 4. Using mysql.connector object to establish a connection to my SQL database
'dtsc691vehicles'
#
---------------------------------------------------------------------------------------------
-------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  F. Incidents Table - 1,500 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when
attempting to insert.

# 7. Closing Database Connection
#
---------------------------------------------------------------------------------------------
-------------------------------

cursor = conn.cursor()
try:
    incidents_data = generate_incidents_data(1500)
    insert_query = "INSERT INTO Incidents (IncidentID, CarID, IncidentDate, Description,
Cost) VALUES (%s, %s, %s, %s, %s)"

    # Inserting data for Owners table using executemany()
    cursor.executemany(insert_query, incidents_data)
```

```
    conn.commit()

    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# In[16]:


# 4. Using mysql.connector object to establish a connection to my SQL database
'dtsc691vehicles'
#
---------------------------------------------------------------------------------------------
-------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  G. ServiceHistory Table - 4,500 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when
attempting to insert.

# 7. Closing Database Connection
#
---------------------------------------------------------------------------------------------
-------------------------------

cursor = conn.cursor()
try:
    service_history_data = generate_service_history_data(4500)
    insert_query = "INSERT INTO ServiceHistory (ServiceID, CarID, ServiceDate, ServiceType,
Cost) VALUES (%s, %s, %s, %s, %s)"

    # Inserting data for Owners table using executemany()
    cursor.executemany(insert_query, service_history_data)
    conn.commit()
```

```
    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# In[17]:


# 4. Using mysql.connector object to establish a connection to my SQL database
'dtsc691vehicles'
#
---------------------------------------------------------------------------------------------
-------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 5. Generating fake data and inserting into respective tables.
#  H. MarketTrends Table - 4,500 records generaeted to be inserted

# 6. In order to handle errors during insertion I use
    # try:
    # except:
    # finally:
# to indicate either that the data inserted correctly or it encountered errors when
attempting to insert.

# 7. Closing Database Connection
#
---------------------------------------------------------------------------------------------
-------------------------------

cursor = conn.cursor()
try:
    market_trends_data = generate_market_trends_data(2000)
    insert_query = "INSERT INTO MarketTrends (TrendID, CarID, Date, AverageSalePrice,
MarketDemand) VALUES (%s, %s, %s, %s, %s)"

    # Inserting data for Owners table using executemany()
    cursor.executemany(insert_query, market_trends_data)
    conn.commit()
```

```
    print("Data inserted successfully.")
except mysql.connector.Error as e:
    print(f"Error inserting data: {e}")
finally:
    # Close cursor and connection
    cursor.close()
    conn.close()


# ### **All of my data was inserted successfully into my SQL database.

# # ----------------------------------- END Python Script
-----------------------------------------------
```

1. Link - [Data Cleaning]
2. Full Code:

```
# ## Python Script Overview
#
# This Python code performs various data cleaning and preprocessing tasks on multiple
dataframes related to automotive data. Here's a summary of what each part of the code does:
#
#    1. Importing Libraries: Imports necessary libraries such as mysql.connector, pandas,
matplotlib.pyplot, LabelEncoder from sklearn.preprocessing, and datetime.
#
#
#    2. Data Retrieval: Retrieves data from MySQL database tables into separate pandas
DataFrames.
#
#
#    3. Handling Missing Values: Checks for missing values in each DataFrame.
#
#
#    4. Outlier Detection and Handling:
#        - Visualizes the distribution of the 'Mileage' column using a box plot.
#        - Filters out outliers from the 'Mileage' column using the 99th percentile.
#        - Filters data based on specified date ranges for the 'OwnershipHistory' table.
#        - Identifies outliers in the 'VehicleCondition' table using the Interquartile Range
(IQR) method.
#
#
#    5. Standardizing Data Formats: Standardizes date columns in several DataFrames.
#
#
#    6. Saving Cleaned Datasets: Saves cleaned DataFrames to CSV files.
#
#
# Overall, this code prepares the data for further analysis and visualization by handling
```

```
missing values, outliers, and standardizing formats. Finally, it saves the cleaned datasets
for future use.
#
# For the code for each of the above components, I will re-iterate with in-line comments
what is mentioned above to identify what code corresponds to which component from above.
Also to keep the context consistent.

# # ----------------------------------- Start Python Script
-----------------------------------------------

# In[1]:


# 1. Importing Necessary Libraries
#
---------------------------------------------------------------------------------------------
-------------------------------

import mysql.connector
import pandas as pd
import matplotlib.pyplot as plt


# The following 45 lines of code pertain to the cleaning of my datasets. By cleaning I am
referring to the handling of
# missing values, addressing outliers, and standardizing my data formats to ensure I have
final 'clean' datasets
# or further analysis and visualization. First we will tackle any missing values.

# In[2]:


# 2A. Connect to database for sunsequent data retrieval
#
---------------------------------------------------------------------------------------------
-------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 2B. Actual data retrieval
#
---------------------------------------------------------------------------------------------
-------------------------------

Cars_df = pd.read_sql_query("SELECT * FROM Cars", conn)
Owners_df = pd.read_sql_query("SELECT * FROM Owners", conn)
```

```
OwnershipHistory_df = pd.read_sql_query("SELECT * FROM OwnershipHistory", conn)
VehicleCondition_df = pd.read_sql_query("SELECT * FROM VehicleCondition", conn)
Features_df = pd.read_sql_query("SELECT * FROM Features", conn)
Incidents_df = pd.read_sql_query("SELECT * FROM Incidents", conn)
ServiceHistory_df = pd.read_sql_query("SELECT * FROM ServiceHistory", conn)
MarketTrends_df = pd.read_sql_query("SELECT * FROM MarketTrends", conn)


# In[4]:


# 3. Handling missing values
#
---------------------------------------------------------------------------------------------
-------------------------------

# Check for missing values in each dataframe
Cars_missing_values = Cars_df.isna().sum()
Owners_missing_values = Owners_df.isna().sum()
OwnershipHistory_missing_values = OwnershipHistory_df.isna().sum()
VehicleCondition_missing_values = VehicleCondition_df.isna().sum()
Features_missing_values = Features_df.isna().sum()
Incidents_missing_values = Incidents_df.isna().sum()
ServiceHistory_missing_values = ServiceHistory_df.isna().sum()
MarketTrends_missing_values = MarketTrends_df.isna().sum()


# In[5]:


Cars_missing_values


# In[6]:


Owners_missing_values


# In[7]:


OwnershipHistory_missing_values


# In[8]:


VehicleCondition_missing_values


# In[9]:
```

```
Features_missing_values


# In[10]:


Incidents_missing_values


# In[11]:


ServiceHistory_missing_values


# In[12]:


MarketTrends_missing_values


# In[13]:


# Based on initial findings above, it appears as though I luckily do not have to deal with any missing values - which
# does make sense based on the simple fact that I generated the data myself and made sure every datapoint was populated
# in each of my tables.


# In[14]:


# There are some points of interest (dataframe columns) that I would like to examine for outliers however.


# In[15]:


# 4. Outlier Detection and Handling
# -----------------------------------------------------------------------------------------------------------------------

# Visualize distribution of 'Mileage' using a box plot
plt.boxplot(Cars_df['Mileage'])
plt.xlabel('Mileage')
plt.title('Box Plot of Mileage')
```

```
plt.show()


# In[16]:


Cars_df.describe()


# In[17]:


# Filtering outliers from the 'Mileage' column

mileage_threshold = Cars_df['Mileage'].quantile(0.99)
Cars_df = Cars_df[Cars_df['Mileage'] <= mileage_threshold]


# In[18]:


Cars_df.describe()


# In[19]:


# For my ownershiphistory data I am not interested in data for records before the year 2020
or after the year 2023.


# In[20]:


# Check min date value
OwnershipHistory_df['SaleDate'].min()


# In[21]:


# Check max date value
OwnershipHistory_df['SaleDate'].max()


# In[22]:


# Create custom start and end date variables

start_date = pd.to_datetime('2020-01-01')
end_date = pd.to_datetime('2024-01-01')
```

```
# In[23]:


# Filtering data based on specified date ranges

OwnershipHistory_df = OwnershipHistory_df[(OwnershipHistory_df['PurchaseDate'] >=
start_date) & (OwnershipHistory_df['PurchaseDate'] <= end_date)]
OwnershipHistory_df = OwnershipHistory_df[(OwnershipHistory_df['SaleDate'] >= start_date) &
(OwnershipHistory_df['SaleDate'] <= end_date)]


# In[24]:


# Check min date value after change
OwnershipHistory_df['SaleDate'].min()


# In[25]:


# Check max date value after change
OwnershipHistory_df['SaleDate'].max()


# In[26]:


# Calculate the quartiles
Q1 = VehicleCondition_df.quantile(0.25)
Q3 = VehicleCondition_df.quantile(0.75)

# Calculate the IQR
IQR = Q3 - Q1

# Define the lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Identify outliers
outliers = ((VehicleCondition_df < lower_bound) | (VehicleCondition_df >
upper_bound)).any(axis=1)

# Display the outliers
print(VehicleCondition_df[outliers])


# In[27]:
```

```
## ^ no outliers


# In[28]:


# Now for the rest of the dataframes, I create individual functions for detecting the
outliers using interquartile range


# In[29]:


def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = ((df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)).any()
    return outliers

outliers_mask = detect_outliers_iqr(Features_df, 'FeatureID')

# Check if any outliers are detected
if outliers_mask.any():
    # Filter the DataFrame to select only the rows that are outliers
    outliers_df = Features_df[outliers_mask]

    # Print the DataFrame containing outliers
    print(outliers_df)
else:
    print("No outliers detected.")


# In[30]:


# Function to detect outliers using Interquartile Range (IQR)
def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers_mask = (df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)
    return outliers_mask

# Detect outliers in Cost column
outliers_mask = detect_outliers_iqr(Incidents_df, 'Cost')

# Filter the DataFrame to select only the rows that are outliers
```

```
outliers_df = Incidents_df[outliers_mask]

# Print the DataFrame containing outliers
print(outliers_df)


# In[31]:


## ^ no outliers


# In[32]:


# Function to detect outliers using Interquartile Range (IQR)
def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers_mask = (df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)
    return outliers_mask

# Detect outliers in Cost column
outliers_mask = detect_outliers_iqr(ServiceHistory_df, 'Cost')

# Filter the DataFrame to select only the rows that are outliers
outliers_df = ServiceHistory_df[outliers_mask]

# Print the DataFrame containing outliers
print(outliers_df)


# In[33]:


## ^ no outliers


# In[34]:


# Function to detect outliers using Interquartile Range (IQR)
def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers_mask = (df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)
```

```
    return outliers_mask

# Detect outliers in AverageSalePrice column
outliers_mask = detect_outliers_iqr(MarketTrends_df, 'AverageSalePrice')

# Filter the DataFrame to select only the rows that are outliers
outliers_df = MarketTrends_df[outliers_mask]

# Print the DataFrame containing outliers
print(outliers_df)


# In[35]:


## ^ no outliers


# My next task is to standardize my data formats

# In[37]:


# 5. Standardize data formats
#   A. Cars table
#
-----------------------------------------------------------------------------------------------
-------------------------------

# For my Cars_df; Convert 'Make', 'Model' to lowercase

Cars_df['Make'] = Cars_df['Make'].str.lower()
Cars_df['Model'] = Cars_df['Model'].str.lower()


# In[38]:


# 5. Standardize data formats
#   B. Owners table
#
-----------------------------------------------------------------------------------------------
-------------------------------

# Owners_df is already standardized


# In[39]:


# 5. Standardize data formats
#   C. OwnershipHistory Table
```

```
#
---------------------------------------------------------------------------------------------
-------------------------------

# For OwnershipHistory_df, I will standardize the date columns

OwnershipHistory_df['PurchaseDate'] = pd.to_datetime(OwnershipHistory_df['PurchaseDate'])
OwnershipHistory_df['SaleDate'] = pd.to_datetime(OwnershipHistory_df['SaleDate'])


# In[40]:


# 5. Standardize data formats
#   D. VehicleCondition Table
#
---------------------------------------------------------------------------------------------
-------------------------------

# VehicleCondition_df is already standardized


# In[41]:


# 5. Standardize data formats
#   E. Features Table
#
---------------------------------------------------------------------------------------------
-------------------------------

# Features_df is already standardized


# In[42]:


# 5. Standardize data formats
#   F. Incidents Table
#
---------------------------------------------------------------------------------------------
-------------------------------

# For my Incidents_df; Convert 'description' to lowercase and also standardize the
IncidentDate

Incidents_df['Description'] = Incidents_df['Description'].str.lower()
Incidents_df['IncidentDate'] = pd.to_datetime(Incidents_df['IncidentDate'])


# In[43]:
```

```
# 5. Standardize data formats
#   G. ServiceHistory Table
#
-------------------------------------------------------------------------------------------
------------------------------

# For ServiceHistoruy data I am going to standardize the data column

ServiceHistory_df['ServiceDate'] = pd.to_datetime(ServiceHistory_df['ServiceDate'])


# In[44]:


# 5. Standardize data formats
#   H. MarketTrends Table
#
-------------------------------------------------------------------------------------------
------------------------------

# For MarketTrends data I am going to standardize the data column

MarketTrends_df['Date'] = pd.to_datetime(MarketTrends_df['Date'])


# Now I have handled mising values, outliers, and standardized my data formats.
# I am ready to re-insert my data into my database and can start in a new notebook for the
analysis and visualization

# In[46]:


# 6. Saving cleaned datasets
#
-------------------------------------------------------------------------------------------
------------------------------

Cars_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\Cars_df.csv', index=False)
Owners_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\Owners_df.csv', index=False)
OwnershipHistory_df.to_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\OwnershipHistory_df.csv', index=False)
VehicleCondition_df.to_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\VehicleCondition_df.csv', index=False)
Features_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\Features_df.csv', index=False)
Incidents_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\Incidents_df.csv', index=False)
```

```
ServiceHistory_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\ServiceHistory_df.csv', index=False)
MarketTrends_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\MarketTrends_df.csv', index=False)


# ### **Now in a new notebook I will re-read the CSVs into python dataframes.
# ### **(Somewhat redundant but allows me to preserve original data)

# # ----------------------------------- END Python Script
-----------------------------------------------
```

1. Link - [Data Cleaning With Encoding]
2. Full Code:

```
# ### Note: I did not end up using the encoded variables or features that I engineered
myself and so the Original 'Data Cleaning.py' file is the version I ran and used for further
analysis. I am including this script in addition in order to showcase that I have the
knowledge required for encoding categorical variables and engineering custom features.
# ###
---------------------------------------------------------------------------------------------
-----------------------------------------------------------------
# ## Python Script Overview
#
# This Python code performs various data cleaning and preprocessing tasks on multiple
dataframes related to automotive data. Here's a summary of what each part of the code does:
#
#    1. Importing Libraries: Imports necessary libraries such as mysql.connector, pandas,
matplotlib.pyplot, LabelEncoder from sklearn.preprocessing, and datetime.
#
#
#    2. Data Retrieval: Retrieves data from MySQL database tables into separate pandas
DataFrames.
#
#
#    3. Handling Missing Values: Checks for missing values in each DataFrame.
#
#
#    4. Outlier Detection and Handling:
#        - Visualizes the distribution of the 'Mileage' column using a box plot.
#        - Filters out outliers from the 'Mileage' column using the 99th percentile.
#        - Filters data based on specified date ranges for the 'OwnershipHistory' table.
#        - Identifies outliers in the 'VehicleCondition' table using the Interquartile Range
(IQR) method.
#
#
#    5. Standardizing Data Formats: Standardizes date columns in several DataFrames.
#
```

```
#
#    6. Encoding Categorical Variables: Encodes categorical variables using label encoding
and one-hot encoding.
#
#
#    7. Feature Engineering:
#        - Calculates new features such as car age, luxury brand indicator, mileage per
year, length of ownership, etc.
#
#
#    8. Saving Cleaned Datasets: Saves cleaned DataFrames to CSV files.
#
#
# Overall, this code prepares the data for further analysis and visualization by handling
missing values, outliers, standardizing formats, encoding categorical variables, and
engineering new features. Finally, it saves the cleaned datasets for future use.
#
# For the code for each of the above components, I will re-iterate with in-line comments
what is mentioned above to identify what code corresponds to which component from above.
Also to keep the context consistent.

# # ----------------------------------- Start Python Script
-----------------------------------------------

# In[1]:


# 1. Importing Necessary Libraries
#
---------------------------------------------------------------------------------------------
-------------------------------

import mysql.connector
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from datetime import datetime


# The following 58 lines of code pertain to the cleaning of my datasets. By cleaning I am
referring to the handling of
# missing values, addressing outliers, standardizing my data formats, encoding my
categorical variables, and engineering
# new features to ensure I have final 'clean' datasets for further analysis and
visualization.
#
# First we will tackle any missing values.

# In[3]:


# 2A. Connect to database for sunsequent data retrieval
```

```
#
---------------------------------------------------------------------------------------------
------------------------------

conn = mysql.connector.connect(
    host='localhost',
    user='paul_walker',
    password='dtsc691root',
    database='dtsc_vehicles'
)


# 2B. Actual data retrieval
#
---------------------------------------------------------------------------------------------
------------------------------

Cars_df = pd.read_sql_query("SELECT * FROM Cars", conn)
Owners_df = pd.read_sql_query("SELECT * FROM Owners", conn)
OwnershipHistory_df = pd.read_sql_query("SELECT * FROM OwnershipHistory", conn)
VehicleCondition_df = pd.read_sql_query("SELECT * FROM VehicleCondition", conn)
Features_df = pd.read_sql_query("SELECT * FROM Features", conn)
Incidents_df = pd.read_sql_query("SELECT * FROM Incidents", conn)
ServiceHistory_df = pd.read_sql_query("SELECT * FROM ServiceHistory", conn)
MarketTrends_df = pd.read_sql_query("SELECT * FROM MarketTrends", conn)


# In[4]:


Owners_df


# In[5]:


# 3. Handling missing values
#
---------------------------------------------------------------------------------------------
------------------------------

# Check for missing values in each dataframe
Cars_missing_values = Cars_df.isna().sum()
Owners_missing_values = Owners_df.isna().sum()
OwnershipHistory_missing_values = OwnershipHistory_df.isna().sum()
VehicleCondition_missing_values = VehicleCondition_df.isna().sum()
Features_missing_values = Features_df.isna().sum()
Incidents_missing_values = Incidents_df.isna().sum()
ServiceHistory_missing_values = ServiceHistory_df.isna().sum()
MarketTrends_missing_values = MarketTrends_df.isna().sum()
```

```
# In[6]:


Cars_missing_values


# In[7]:


Owners_missing_values


# In[8]:


OwnershipHistory_missing_values


# In[9]:


VehicleCondition_missing_values


# In[10]:


Features_missing_values


# In[11]:


Incidents_missing_values


# In[12]:


ServiceHistory_missing_values


# In[13]:


MarketTrends_missing_values


# In[14]:


# Based on initial findings above, it appears as though I luckily do not have to deal with
```

```
any missing values - which
# does make sense based on the simple fact that I generated the data myself and made sure
every datapoint was populated
# in each of my tables.


# In[15]:


# There are some points of interest (dataframe columns) that I would like to examine for
outliers however.


# In[16]:


# 4. Outlier Detection and Handling
#
---------------------------------------------------------------------------------------------
-------------------------------

# Visualize distribution of 'Mileage' using a box plot
plt.boxplot(Cars_df['Mileage'])
plt.xlabel('Mileage')
plt.title('Box Plot of Mileage')
plt.show()


# In[17]:


Cars_df.describe()


# In[18]:


# Filtering outliers from the 'Mileage' column

mileage_threshold = Cars_df['Mileage'].quantile(0.99)
Cars_df = Cars_df[Cars_df['Mileage'] <= mileage_threshold]


# In[19]:


Cars_df.describe()


# In[20]:
```

```
# For my ownershiphistory data I am not interested in data for records before the year 2020
or after the year 2023.


# In[21]:


# Check min date value
OwnershipHistory_df['SaleDate'].min()


# In[22]:


# Check max date value
OwnershipHistory_df['SaleDate'].max()


# In[23]:


# Create custom start and end date variables

start_date = pd.to_datetime('2020-01-01')
end_date = pd.to_datetime('2024-01-01')


# In[24]:


# Filtering data based on specified date ranges

OwnershipHistory_df = OwnershipHistory_df[(OwnershipHistory_df['PurchaseDate'] >=
start_date) & (OwnershipHistory_df['PurchaseDate'] <= end_date)]
OwnershipHistory_df = OwnershipHistory_df[(OwnershipHistory_df['SaleDate'] >= start_date) &
(OwnershipHistory_df['SaleDate'] <= end_date)]


# In[25]:


# Check min date value after change

OwnershipHistory_df['SaleDate'].min()


# In[26]:


# Check max date value after change

OwnershipHistory_df['SaleDate'].max()
```

```
# In[27]:


# Calculate the quartiles
Q1 = VehicleCondition_df.quantile(0.25)
Q3 = VehicleCondition_df.quantile(0.75)

# Calculate the IQR
IQR = Q3 - Q1

# Define the lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Identify outliers
outliers = ((VehicleCondition_df < lower_bound) | (VehicleCondition_df >
upper_bound)).any(axis=1)

# Display the outliers
print(VehicleCondition_df[outliers])


# In[28]:


## ^ no outliers


# In[29]:


# Now for the rest of the dataframes, I create individual functions for detecting the
outliers using interquartile range


# In[30]:


def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = ((df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)).any()
    return outliers

outliers_mask = detect_outliers_iqr(Features_df, 'FeatureID')

# Check if any outliers are detected
```

```python
if outliers_mask.any():
    # Filter the DataFrame to select only the rows that are outliers
    outliers_df = Features_df[outliers_mask]

    # Print the DataFrame containing outliers
    print(outliers_df)
else:
    print("No outliers detected.")


# In[31]:


# Function to detect outliers using Interquartile Range (IQR)
def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers_mask = (df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)
    return outliers_mask

# Detect outliers in Cost column
outliers_mask = detect_outliers_iqr(Incidents_df, 'Cost')

# Filter the DataFrame to select only the rows that are outliers
outliers_df = Incidents_df[outliers_mask]

# Print the DataFrame containing outliers
print(outliers_df)


# In[32]:


## ^ no outliers


# In[33]:


# Function to detect outliers using Interquartile Range (IQR)
def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers_mask = (df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)
    return outliers_mask
```

```
# Detect outliers in Cost column
outliers_mask = detect_outliers_iqr(ServiceHistory_df, 'Cost')

# Filter the DataFrame to select only the rows that are outliers
outliers_df = ServiceHistory_df[outliers_mask]

# Print the DataFrame containing outliers
print(outliers_df)


# In[34]:


## ^ no outliers


# In[35]:


# Function to detect outliers using Interquartile Range (IQR)
def detect_outliers_iqr(df, feature_col):
    Q1 = df[feature_col].quantile(0.25)
    Q3 = df[feature_col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers_mask = (df[feature_col] < lower_bound) | (df[feature_col] > upper_bound)
    return outliers_mask

# Detect outliers in AverageSalePrice column
outliers_mask = detect_outliers_iqr(MarketTrends_df, 'AverageSalePrice')

# Filter the DataFrame to select only the rows that are outliers
outliers_df = MarketTrends_df[outliers_mask]

# Print the DataFrame containing outliers
print(outliers_df)


# In[36]:


## ^ no outliers


# My next task is to standardize my data formats

# In[38]:


# 5. Standardize data formats
#   A. Cars table
```

```
#
----------------------------------------------------------------------------------------------
-------------------------------

# For my Cars_df; Convert 'Make', 'Model' to lowercase

Cars_df['Make'] = Cars_df['Make'].str.lower()
Cars_df['Model'] = Cars_df['Model'].str.lower()


# In[39]:


# 5. Standardize data formats
#   B. Owners table
#
----------------------------------------------------------------------------------------------
-------------------------------

# Owners_df is already standardized


# In[40]:


# 5. Standardize data formats
#   C. OwnershipHistory Table
#
----------------------------------------------------------------------------------------------
-------------------------------

# For OwnershipHistory_df, I will standardize the date columns

OwnershipHistory_df['PurchaseDate'] = pd.to_datetime(OwnershipHistory_df['PurchaseDate'])
OwnershipHistory_df['SaleDate'] = pd.to_datetime(OwnershipHistory_df['SaleDate'])


# In[41]:


# 5. Standardize data formats
#   D. VehicleCondition Table
#
----------------------------------------------------------------------------------------------
-------------------------------

# VehicleCondition_df is already standardized


# In[42]:
```

```
# 5. Standardize data formats
#   E. Features Table
#
-------------------------------------------------------------------------------------------
------------------------------

# Features_df is already standardized


# In[43]:


# 5. Standardize data formats
#   F. Incidents Table
#
-------------------------------------------------------------------------------------------
------------------------------

# For my Incidents_df; Convert 'description' to lowercase and also standardize the
IncidentDate

Incidents_df['Description'] = Incidents_df['Description'].str.lower()
Incidents_df['IncidentDate'] = pd.to_datetime(Incidents_df['IncidentDate'])


# In[44]:


# 5. Standardize data formats
#   G. ServiceHistory Table
#
-------------------------------------------------------------------------------------------
------------------------------

# For ServiceHistoruy data I am going to standardize the data column

ServiceHistory_df['ServiceDate'] = pd.to_datetime(ServiceHistory_df['ServiceDate'])


# In[45]:


# 5. Standardize data formats
#   H. MarketTrends Table
#
-------------------------------------------------------------------------------------------
------------------------------

# For MarketTrends data I am going to standardize the data column

MarketTrends_df['Date'] = pd.to_datetime(MarketTrends_df['Date'])
```

```
# Now for encoding my categorical variables.

# In[47]:


# 6. Encoding Categorical Variables
#   A. Cars table
#
-----------------------------------------------------------------------------------------------------------------------

# In Cars table;
# 1. EngineType - values: ('2-Cylinder', '3-Cylinder', '4-Cylinder', '5-Cylinder',
'6-Cylinder', '8-Cylinder','10-Cylinder+')
# 2. TransmissionType - values: ('Automatic', 'Manual', 'CVT')
# 3. FuelType - values: ('Gasoline', 'Diesel', 'Hybrid', 'Electric')

# Label Encoding for 'EngineType' in Cars_df
label_encoder = LabelEncoder()
Cars_df['EngineType_encoded'] = label_encoder.fit_transform(Cars_df['EngineType'])

# One-Hot Encoding for'TransmissionType' in Cars_df
Cars_df = pd.get_dummies(Cars_df, columns=['TransmissionType'], drop_first=True)

# One-Hot Encoding for 'FuelType' in Cars_df
Cars_df = pd.get_dummies(Cars_df, columns=['FuelType'], drop_first=True)
Cars_df


# In[48]:


# 6. Encoding Categorical Variables
#   B. VehicleCondition table
#
-----------------------------------------------------------------------------------------------------------------------

# In VehicleCondition table;
# 1. OverallCondition - values: ('Excellent', 'Good', 'Fair', 'Poor')
# 2. ExteriorCondition - values: ('Clean', 'Minor Scratches', 'Dents', 'Needs Repairs')
# 3. InteriorCondition - values: ('Clean', 'Minor Wear', 'Torn Upholstery', 'Needs
Cleaning')

# Label Encoding for various conditions in VehicleHistory_df
VehicleCondition_df['OverallCondition_encoded'] =
label_encoder.fit_transform(VehicleCondition_df['OverallCondition'])
VehicleCondition_df['ExteriorCondition_encoded'] =
label_encoder.fit_transform(VehicleCondition_df['ExteriorCondition'])
VehicleCondition_df['InteriorCondition_encoded'] =
label_encoder.fit_transform(VehicleCondition_df['InteriorCondition'])
```

```
VehicleCondition_df


# In[49]:


# 6. Encoding Categorical Variables
#   C. Features table
#
-------------------------------------------------------------------------------------------
------------------------------

# In Features table;
# 1. FeatureName - values: ('Air Conditioning', 'Power Windows', 'ABS', 'Cruise Control',
'Bluetooth', 'Backup Camera')

# Label Encoding feature name in Features_df
Features_df['FeatureName_encoded'] = label_encoder.fit_transform(Features_df['FeatureName'])
Features_df


# In[50]:


# 6. Encoding Categorical Variables
#   D. ServiceHistory table
#
-------------------------------------------------------------------------------------------
------------------------------

# In ServiceHistory table;
# 1. ServiceType - values: ('Oil Change', 'Brake Inspection', 'Tire Rotation', 'Engine
Tune-up')

# Label Encoding for ServiceType in ServiceHistory_df
ServiceHistory_df['ServiceType_encoded'] =
label_encoder.fit_transform(ServiceHistory_df['ServiceType'])
ServiceHistory_df


# Now for engineering new features.

# In[52]:


# 7. Feature Engineering
#   A. Cars Table:
#
-------------------------------------------------------------------------------------------
------------------------------

# Calculate the age of the car based on the current year and the 'Year' column.
```

```
current_year = datetime.now().year
Cars_df['Age'] = current_year - Cars_df['Year']


# Create a binary feature indicating whether the car is a luxury brand or not based on the
'Make' column.
luxury_brands = ['Mercedes', 'BMW', 'Audi', 'Lexus', 'Porsche', 'Jaguar', 'Infiniti',
'Acura', 'Cadillac', 'Lincoln']
Cars_df['IsLuxury'] = Cars_df['Make'].apply(lambda x: 1 if any(brand in x for brand in
luxury_brands) else 0)


# Calculate the mileage per year for each car.
Cars_df['MileagePerYear'] = Cars_df['Mileage'] / Cars_df['Age']

Cars_df.head()


# In[53]:


# 7. Feature Engineering
#   B. Owners Table:
#
-----------------------------------------------------------------------------------------------
-------------------------------

# Calculate the length of ownership for each owner by subtracting PurchaseDate from SaleDate
Owners_df['LengthOfOwnership'] = (OwnershipHistory_df['SaleDate'] -
OwnershipHistory_df['PurchaseDate']).dt.days

# Determine the number of cars each owner has owned
car_counts = OwnershipHistory_df['OwnerID'].value_counts().rename('NumCarsOwned')
Owners_df = Owners_df.merge(car_counts, left_on='OwnerID', right_index=True, how='left')

# Encode the contact information to identify patterns such as phone number format or email
domain
# For simplicity, let's just check if the contact info contains an email address or phone
number
Owners_df['HasEmail'] = Owners_df['ContactInfo'].str.contains('@').astype(int)
Owners_df['HasPhoneNumber'] =
Owners_df['ContactInfo'].str.contains(r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b').astype(int)

Owners_df.head()


# In[54]:


# 7. Feature Engineering
#   C. OwnershipHistory Table:
#
```

```
-----------------------------------------------------------------------------------------
------------------------------

# Calculate the duration of ownership for each entry
OwnershipHistory_df['OwnershipDuration'] = (OwnershipHistory_df['SaleDate'] -
OwnershipHistory_df['PurchaseDate']).dt.days

# Calculate the percentage change in sale price for each transaction
OwnershipHistory_df['SalePriceChange'] =
OwnershipHistory_df.groupby('CarID')['SalePrice'].pct_change()

# Fill NaN values resulting from the pct_change operation with 0, as the first transaction
won't have a previous value to compare to
OwnershipHistory_df['SalePriceChange'].fillna(0, inplace=True)

OwnershipHistory_df.head()


# In[55]:


# 7. Feature Engineering
#   D. VehicleCondition Table:
#
-----------------------------------------------------------------------------------------
------------------------------

# Define a mapping of condition metrics to scores
condition_mapping = {
    'Excellent': 5,
    'Good': 4,
    'Fair': 3,
    'Poor': 2,
    'Very Poor': 1
}

# Calculate an overall condition score based on the given condition metrics
def calculate_overall_condition(row):
    overall_condition_score = 0
    condition_metrics = [row['OverallCondition'], row['ExteriorCondition'],
row['InteriorCondition']]
    for condition in condition_metrics:
        overall_condition_score += condition_mapping.get(condition, 0)
    return overall_condition_score / len(condition_metrics)

VehicleCondition_df['OverallConditionScore'] =
VehicleCondition_df.apply(calculate_overall_condition, axis=1)

VehicleCondition_df.head()


# In[56]:
```

```
# 7. Feature Engineering
#   E. Features Table:
#
-----------------------------------------------------------------------------------------------
-------------------------------

# Calculate the number of features for each car
num_features_per_car = Features_df.groupby('CarID').size().rename('NumFeatures')

# Merge the calculated number of features with the main DataFrame
Features_df = Features_df.merge(num_features_per_car, left_on='CarID', right_index=True,
how='left')

Features_df.head()


# In[57]:


# 7. Feature Engineering
#   F. Incidents Table:
#
-----------------------------------------------------------------------------------------------
-------------------------------

# Calculate the average cost per incident
average_cost_per_incident =
Incidents_df.groupby('CarID')['Cost'].mean().rename('AvgCostPerIncident')

# Merge the calculated average cost per incident with the main DataFrame
Incidents_df = Incidents_df.merge(average_cost_per_incident, left_on='CarID',
right_index=True, how='left')

Incidents_df.head()


# In[58]:


# 7. Feature Engineering
#   G. ServiceHistory Table:
#
-----------------------------------------------------------------------------------------------
-------------------------------

# Calculate the frequency of services per car
service_frequency_per_car =
ServiceHistory_df.groupby('CarID').size().rename('ServiceFrequency')

# Calculate the total cost of services per car
```

```
total_cost_of_services_per_car =
ServiceHistory_df.groupby('CarID')['Cost'].sum().rename('TotalCostOfServices')

# Merge the calculated features with the main DataFrame
ServiceHistory_df = ServiceHistory_df.merge(service_frequency_per_car, left_on='CarID',
right_index=True, how='left')
ServiceHistory_df = ServiceHistory_df.merge(total_cost_of_services_per_car, left_on='CarID',
right_index=True, how='left')

ServiceHistory_df.head()


# In[59]:


# 7. Feature Engineering
#   H. MarketTrends Table:
#
---------------------------------------------------------------------------------------------
-------------------------------

# Sort the DataFrame by CarID and Date
MarketTrends_df.sort_values(by=['CarID', 'Date'], inplace=True)

# Calculate the percentage change in average sale price compared to the previous period
MarketTrends_df['AvgSalePriceChange'] =
MarketTrends_df.groupby('CarID')['AverageSalePrice'].pct_change()

# Create a feature indicating whether the market demand is increasing or decreasing
MarketTrends_df['DemandChange'] = pd.cut(MarketTrends_df['MarketDemand'].diff(),
bins=[float('-inf'), 0, float('inf')], labels=['Decreasing', 'Increasing'])

MarketTrends_df.head()


# Now I have handled, mising values, outliers, standardized my data formats, encoded my
categorical variables, and engineered
# new features. I am ready to re-insert my data into my database and can start in a new
notebook for the
# analysis and visualization.

# In[61]:


# 8. Saving cleaned datasets
#
---------------------------------------------------------------------------------------------
-------------------------------

# Cars_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\Cars_dfV2.csv', index=False)
# Owners_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
```

```
University\DTSC 691 - Capstone II\Clean Datasets\Owners_dfV2.csv', index=False)
# OwnershipHistory_df.to_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\OwnershipHistory_dfV2.csv', index=False)
# VehicleCondition_df.to_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\VehicleCondition_dfV2.csv', index=False)
# Features_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\Features_dfV2.csv', index=False)
# Incidents_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\Incidents_dfV2.csv', index=False)
# ServiceHistory_df.to_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\ServiceHistory_dfV2.csv', index=False)
# MarketTrends_df.to_csv(r'P:\Users\paulj\Desktop\Important Documentation\Education\Eastern
University\DTSC 691 - Capstone II\Clean Datasets\MarketTrends_dfV2.csv', index=False)


# ### **As mentioned at beginning of file, I am not utilizing these versions of my cleaned
datasets and so that is why all of the ".to_csv()" functions have been commented out.
```

1. Link - Data analysis
2. Full Code:

```
# ## Python Script Overview
#
# The provided Python code conducts a comprehensive data analysis on
several datasets related to vehicle information. Here's a summary and
commentary on each section:
#
#    1. Importing Libraries: Imports necessary libraries such as pandas and
numpy, matplotlib.pyplot, and various statistical models from scipy.
#
#
#    2. Data Loading:
#        - Dataframes for various datasets like Cars, Owners, Ownership
History, etc., are loaded from CSV files into Pandas DataFrames.
#
#
#    3. Data Summary:
#        - The .head() method is used to display the first few rows of each
DataFrame for initial inspection.
#        - The .describe() method is applied to each DataFrame to get
summary statistics like count, mean, std, min, max, etc.
```

```
#
#
#    4. Correlation Analysis:
#        - Pearson correlation coefficients are calculated for each pair of variables in each DataFrame using the .corr() method.
#        - Correlation matrices are printed for each DataFrame to understand the relationships between variables.
#
#
#    5. Hypothesis Testing:
#        - Several hypothesis tests are conducted to analyze different aspects of the data.
#            - For example, the effect of mileage on sale price is tested using Pearson correlation and linear regression.
#        - ANOVA tests are performed to analyze differences in average sale price based on car make, vehicle condition, and fuel type.
#        - Chi-square test of independence and logistic regression are used to examine the association between incidents and market demand.
#        - Time-series analysis is conducted to identify trends in average sale price over time.
#
#
#    6. Data Manipulation:
#        - Dataframes are merged, cleaned, and processed as needed for hypothesis testing and analysis.
#            - For instance, missing values are handled, and new columns like 'Age' and 'MPY' (Miles Per Year) are calculated.
#
#
#    7. Data Visualization:
#        - Matplotlib is used to visualize data trends and relationships, such as plotting average sale price over time.
#
#
# Overall, the code provides a thorough exploration of the datasets, conducts various statistical analyses, and generates visualizations to gain insights into different aspects of the vehicle data.
#
# For the code for each of the above components, I will re-iterate with in-line comments what is mentioned above to identify what code corresponds to which component from above. Also to keep the context consistent.

# # ---------------------------------- Start Python Script
```

```
-----------------------------------------------


# In[1]:


# 1. Importing Necessary Libraries
#
---------------------------------------------------------------------------
------------------------------------------------

import pandas as pd
import numpy as np

import statsmodels.api as sm
import matplotlib.pyplot as plt

from scipy.stats import chi2_contingency
from scipy.stats import f_oneway
from scipy.stats import pearsonr


# In[2]:


# 2. Data loading - using pandas function ".read_csv()" since in my prior
script I retrieved the data from my SQL database,
# cleaned it, and saved it to .csv files on my own machine.
#
---------------------------------------------------------------------------
------------------------------------------------

Cars_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Cars_df.csv')

Owners_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Owners_df.csv')

OwnershipHistory_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\OwnershipHistory_df.csv')
```

```
VehicleCondition_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\VehicleCondition_df.csv')

Features_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Features_df.csv')

Incidents_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Incidents_df.csv')

ServiceHistory_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\ServiceHistory_df.csv')

MarketTrends_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\MarketTrends_df.csv')


# In[3]:


# 3. Data summaries 1
#   A. Cars_df
#
---------------------------------------------------------------------------
------------------------------------------------

Cars_df.head(5)


# In[4]:


# 3. Data summaries 1
#   B. Owners_df
#
---------------------------------------------------------------------------
------------------------------------------------

Owners_df.head(5)
```

```
# In[5]:


# 3. Data summaries 1
#   C. OwnershipHistory_df
#
-----------------------------------------------------------------------------
------------------------------------------------

OwnershipHistory_df.head(5)


# In[6]:


# 3. Data summaries 1
#   D. VehicleCondition_df
#
-----------------------------------------------------------------------------
------------------------------------------------

VehicleCondition_df.head(5)


# In[7]:


# 3. Data summaries 1
#   E. Features_df
#
-----------------------------------------------------------------------------
------------------------------------------------

Features_df.head(5)


# In[8]:


# 3. Data summaries 1
#   F. Incidents_df
```

```
#
---------------------------------------------------------------------------
-----------------------------------------------

Incidents_df.head(5)


# In[9]:


# 3. Data summaries 1
#   G. ServiceHistory_df
#
---------------------------------------------------------------------------
-----------------------------------------------

ServiceHistory_df.head(5)


# In[10]:


# 3. Data summaries 1
#   H. MarketTrends_df
#
---------------------------------------------------------------------------
-----------------------------------------------

MarketTrends_df.head(5)


# In[11]:


# 3. Data summaries 2
#   A. Cars_df
#
---------------------------------------------------------------------------
-----------------------------------------------

summary_stats = Cars_df.describe()
print(summary_stats)
```

```
# In[12]:


# 3. Data summaries 2
#   B. OwnershipHistory_df
#
---------------------------------------------------------------------------
-----------------------------------------------

summary_stats = OwnershipHistory_df.describe()
print(summary_stats)


# In[13]:


# 3. Data summaries 2
#   C. Incidents_df
#
---------------------------------------------------------------------------
-----------------------------------------------

summary_stats = Incidents_df.describe()
print(summary_stats)


# In[14]:


# 3. Data summaries 2
#   D. ServiceHistory_df
#
---------------------------------------------------------------------------
-----------------------------------------------

summary_stats = ServiceHistory_df.describe()
print(summary_stats)


# In[15]:
```

```
# 4. Pearson Correlation Coefficient
#   A. Cars_df
#
---------------------------------------------------------------------------
-----------------------------------------------

Cars_corr = Cars_df.corr(method = 'pearson')
print(Cars_corr)


# 1. CarID and Year: There is a very weak negative correlation (-0.018717),
suggesting that there is virtually no linear relationship between the car's
ID and the year it was made.
#
# 2. CarID and Mileage: There is a very weak negative correlation
(-0.030113), suggesting that there is virtually no linear relationship
between the car's ID and its mileage.
#
# 3. Year and Mileage: There is a very weak negative correlation
(-0.010199), indicating that there is virtually no linear relationship
between the year the car was made and its mileage.

# In[16]:


# 4. Pearson Correlation Coefficient
#   B. Owners_df
#
---------------------------------------------------------------------------
-----------------------------------------------

Owners_corr = Owners_df.corr(method = 'pearson')
print(Owners_corr)


# In[17]:


# 4. Pearson Correlation Coefficient
#   C. OwnershipHistory_df
#
---------------------------------------------------------------------------
-----------------------------------------------
```

```
OwnershipHistory_corr = OwnershipHistory_df.corr(method = 'pearson')
print(OwnershipHistory_corr)


# 1. OwnershipID and CarID: Very weak negative correlation (-0.097907),
indicating no meaningful linear relationship.
#
# 2. OwnershipID and OwnerID: Very weak positive correlation (0.005042),
indicating no meaningful linear relationship.
#
# 3. OwnershipID and SalePrice: Weak negative correlation (-0.029520),
indicating a negligible linear relationship.
#
# 4. CarID and OwnerID: Very weak positive correlation (0.005434),
indicating no meaningful linear relationship.
#
# 5. CarID and SalePrice: Weak negative correlation (-0.034504), suggesting
a very slight tendency for cars with higher IDs to have lower sale prices,
but the relationship is not strong.
#
# 6. OwnerID and SalePrice: Weak negative correlation (-0.017669),
indicating a very slight tendency for owners with higher IDs to have
transactions with lower sale prices, which is likely not a meaningful
relationship.

# In[18]:


# 4. Pearson Correlation Coefficient
#   D. VehicleCondition_df
#
---------------------------------------------------------------------------
----------------------------------------------

VehicleCondition_corr = VehicleCondition_df.corr(method = 'pearson')
print(VehicleCondition_corr)


# In[19]:


# 4. Pearson Correlation Coefficient
```

```
#   E. Features_df
#
--------------------------------------------------------------------------------------------------------------------------
Features_corr = Features_df.corr(method = 'pearson')
print(Features_corr)


# In[20]:


# 4. Pearson Correlation Coefficient
#   F. Incidents_df
#
--------------------------------------------------------------------------------------------------------------------------
Incidents_corr = Incidents_df.corr(method = 'pearson')
print(Incidents_corr)


# 1. IncidentID and CarID have a very weak positive correlation (0.055759),
which is negligible.
#
# 2. IncidentID and Cost have a very weak negative correlation (-0.001110),
which suggests no meaningful relationship.
#
# 3. CarID and Cost also have a very weak negative correlation (-0.055861),
indicating no significant relationship between the car's ID and the cost
associated with its incidents.

# In[21]:


# 4. Pearson Correlation Coefficient
#   G. ServiceHistory_df
#
--------------------------------------------------------------------------------------------------------------------------
ServiceHistory_corr = ServiceHistory_df.corr(method = 'pearson')
print(ServiceHistory_corr)
```

```
# 1. ServiceID and CarID have a very weak positive correlation (0.013296),
which is negligible.
#
# 2. ServiceID and Cost have a very weak negative correlation (-0.006288),
suggesting no significant relationship.
#
# 3. CarID and Cost have a very weak negative correlation (-0.008768),
which is also negligible.

# In[22]:


# 4. Pearson Correlation Coefficient
#   H. MarketTrends_df
#
---------------------------------------------------------------------------
-----------------------------------------------

MarketTrends_corr = MarketTrends_df.corr(method = 'pearson')
print(MarketTrends_corr)


# 1. TrendID and CarID have a very weak positive correlation (0.016518),
which is negligible.
#
# 2. TrendID and AverageSalePrice have a very weak negative correlation
(-0.019498), indicating no significant relationship.
#
# 3. TrendID and MarketDemand have a weak positive correlation (0.010106),
which is very slight.
#
# 4. CarID and AverageSalePrice have a very weak negative correlation
(-0.015434), suggesting no meaningful relationship.
#
# 5. CarID and MarketDemand have a very weak positive correlation
(0.014181), which is negligible.
#
# 6. AverageSalePrice and MarketDemand have a very weak negative
correlation (-0.003404), indicating no significant relationship.

# In all datasets, the correlations are weak, suggesting that the
```

```
identifiers (like IncidentID, ServiceID, TrendID, CarID) have no meaningful
linear relationship with the other variables such as costs or market
trends. This is expected because identifiers are usually arbitrarily
assigned and should not logically correlate with these variables.

# Now I will create various correlation matrices to examine the
relationships of the variables in each dataframe

# In[23]:


# 4. Correlation Matrices
#
---------------------------------------------------------------------------
-----------------------------------------------

# Correlation matrix for Cars_df
print("Correlation Analysis for Cars_df:")
print(Cars_df.corr())

# Correlation matrix for Owners_df
print("\nCorrelation Analysis for Owners_df:")
print(Owners_df.corr())

# Correlation matrix for OwnershipHistory_df
print("\nCorrelation Analysis for OwnershipHistory_df:")
print(OwnershipHistory_df.corr())

# Correlation matrix for VehicleCondition_df
print("\nCorrelation Analysis for VehicleCondition_df:")
print(VehicleCondition_df.corr())

# Correlation matrix for Features_df
print("\nCorrelation Analysis for Features_df:")
print(Features_df.corr())

# Correlation matrix for Incidents_df (if applicable)
print("\nCorrelation Analysis for Incidents_df:")
print(Incidents_df.corr())

# Correlation matrix for ServiceHistory_df
print("\nCorrelation Analysis for ServiceHistory_df:")
print(ServiceHistory_df.corr())
```

```
# Correlation matrix for MarketTrends_df
print("\nCorrelation Analysis for MarketTrends_df:")
print(MarketTrends_df.corr())


# ## Analysis of results:

# 1. Cars_df: No significant correlation between CarID, Year, and Mileage.
#
# 2. Owners_df: No significant correlation between OwnerID and CarID.
#
# 3. OwnershipHistory_df: Very weak correlations among OwnershipID, CarID,
OwnerID, and SalePrice.
#
# 4. VehicleCondition_df: No information on correlations other than a
perfect correlation of ConditionID with itself.
#
# 5. Features_df: No significant correlation between FeatureID and CarID.
#
# 6. Incidents_df: Very weak correlations among IncidentID, CarID, and
Cost.
#
# 7. ServiceHistory_df: Very weak correlations among ServiceID, CarID, and
Cost.
#
# 8. MarketTrends_df: Very weak correlations among TrendID, CarID,
AverageSalePrice, and MarketDemand.

# Overall, the identifiers (like IDs) show no significant correlations with
other variables, which is expected. Other variables like Cost, SalePrice,
and Market Demand also exhibit very weak correlations with identifiers and
each other, suggesting that they are not linearly related within these
datasets.

# Now for various hypothesis testing analyses using the scipy library

# In[25]:



# 5. Hypothesis Testing - Pearson Correlation and Linear Regression
# 6. Data Manipulation
#
```

```
---------------------------------------------------------------------------
------------------------------------------------

# Hpothesis Test 1: Effect of Mileage on Sale Price
    # Null Hypothesis: There is no significant correlation between the
mileage of a car and its sale price.
    # Alternative Hypothesis: There is a significant correlation between
the mileage of a car and its sale price.
    # Test: Pearson correlation coefficient and linear regression analysis


# In[26]:


# Currently my Cars_df does not have information on the sales prices for
the various vehicles, so I need to
# merge my Cars_df with my OwnershipHistory_df to obtain the sales prices.


# In[27]:


data_df = pd.merge(Cars_df, OwnershipHistory_df[['CarID', 'SalePrice']],
on='CarID', how='left')
data_df.head(5)


# In[28]:


# there are some NaN in SalePrice due to gaps in the OwnershipHistory with
respect to the CarID. I will replace the NaN
# values with the average of the Non-NaN values in SalePrice

average_sale_price = data_df['SalePrice'].mean()
data_df['SalePrice'].fillna(average_sale_price, inplace=True)
data_df.head(5)


# In[29]:


# Calculate Pearson correlation coefficient
```

```
pearson_corr, pearson_p_value = pearsonr(data_df['Mileage'],
data_df['SalePrice'])
print("Pearson Correlation Coefficient:", pearson_corr)
print("P-value:", pearson_p_value)

# Perform linear regression analysis
X = sm.add_constant(data_df['Mileage']) # Adding constant term
y = data_df['SalePrice']
model = sm.OLS(y, X).fit()
print(model.summary())


# ### Hypothesis Test 1 Interpretation:
#
#
# Pearson Correlation Coefficient: The Pearson correlation coefficient
measures the strength and direction of the linear relationship between two
variables. In this case, the coefficient is approximately 0.021, indicating
a very weak positive correlation between mileage and sale price. However,
it's important to note that this correlation is close to zero, suggesting
little to no linear relationship between the two variables.
#
#
# P-value: The p-value associated with the correlation coefficient is
approximately 0.267. This p-value represents the probability of observing
the data given that the null hypothesis (no correlation) is true. Since the
p-value is greater than the conventional significance level of 0.05, we
fail to reject the null hypothesis. This suggests that there is
insufficient evidence to conclude that there is a significant correlation
between mileage and sale price.
#
#
# Linear Regression Analysis: The linear regression model further examines
the relationship between mileage and sale price by estimating the
coefficients of a linear equation (SalePrice = intercept + slope *
Mileage). The coefficient for the 'Mileage' variable is approximately
0.0042, indicating that for each unit increase in mileage, the predicted
change in sale price is very small (0.0042 units), holding all other
variables constant.
#


# In[30]:
```

```
# 5. Hypothesis Testing - One Way ANOVA
# 6. Data Manipulation
#
----------------------------------------------------------------------------
------------------------------------------------

# Hpothesis Test 2: Difference in Average Sale Price between Different Car
Makes:
    # Null Hypothesis: There is no significant difference in the average
sale price between different car makes.
    # Alternative Hypothesis: There is a significant difference in the
average sale price between different car makes.
    # Test: One-way ANOVA


# In[31]:


# I can use my merged dataframe again for these analysis.

# Create a dictionary to store sale prices for each car make
sale_prices_by_make = {}
for make, group in data_df.groupby('Make'):
    sale_prices_by_make[make] = group['SalePrice']

# Perform One-way ANOVA test
f_statistic, p_value = f_oneway(*sale_prices_by_make.values())

# Print results
print("F-statistic:", f_statistic)
print("P-value:", p_value)

# Interpret results
if p_value < 0.05:
    print("Reject null hypothesis: There is a significant difference in
average sale price between different car makes.")
else:
    print("Fail to reject null hypothesis: There is no significant
difference in average sale price between different car makes.")


# ### Hypothesis Test 2 Interpretation:
```

```
#
#
# F-statistic: The F-statistic is approximately 1.071, which is a measure
of the variation between the group means relative to the variation within
the groups.
#
#
# P-value: The p-value associated with the F-statistic is approximately
0.333. This p-value represents the probability of observing the data given
that the null hypothesis (no difference in average sale price between
different car makes) is true.
#
#
# Overall, since the p-value (0.333) is greater than the chosen
significance level (e.g., 0.05), we fail to reject the null hypothesis.
This means that there is insufficient evidence to conclude that there is a
significant difference in average sale price between different car makes.

# In[32]:


# 5. Hypothesis Testing - One Way ANOVA
# 6. Data Manipulation
#
---------------------------------------------------------------------------
------------------------------------------------

# Hpothesis Test 3: Impact of Vehicle Condition on Sale Price:
    # Null Hypothesis: There is no significant difference in the sale price
of cars with different overall conditions.
    # Alternative Hypothesis: There is a significant difference in the sale
price of cars with different overall conditions.
    # Test: One-way ANOVA


# In[33]:


# Again i will need to merge a few of my dataframes in order to perform
this test. I need to merge my OwnershipHistory_df
# with my VehicleCondition_df

data_df2 = pd.merge(data_df, VehicleCondition_df[['CarID',
```

```
'OverallCondition']], on='CarID', how='left')
data_df2


# In[34]:


# there are some NaN in OverallCondition due to gaps in the
VehicleCondition with respect to the CarID. I will drop rows
# with NaN values as I require the OverallCondition information in order to
perform my test.


# In[35]:


data_df2 = data_df2.dropna()
data_df2


# In[36]:


# Perform ANOVA
# Create a dictionary to store sale prices for each vehicle condition
sale_prices_by_condition = {}
for condition, group in data_df2.groupby('OverallCondition'):
    sale_prices_by_condition[condition] = group['SalePrice']

# Perform ANOVA test
f_statistic, p_value = f_oneway(*sale_prices_by_condition.values())

# Print results
print("F-statistic:", f_statistic)
print("P-value:", p_value)

# Interpret results
if p_value < 0.05:
    print("Reject null hypothesis: There is a significant impact of vehicle
condition on sale price.")
else:
    print("Fail to reject null hypothesis: There is no significant impact
of vehicle condition on sale price.")
```

```
# ### Hypothesis Test 3 Interpretation:
#
#
# F-statistic: The F-statistic is approximately 0.177. This statistic measures the variation in sale prices between different vehicle conditions relative to the variation within each vehicle condition group.
#
#
# P-value: The p-value associated with the F-statistic is approximately 0.912. This p-value represents the probability of observing the data given that the null hypothesis (no significant impact of vehicle condition on sale price) is true.
#
#
# Overall, since the p-value (0.912) is much greater than the chosen significance level (e.g., 0.05), we fail to reject the null hypothesis. This means that there is insufficient evidence to conclude that there is a significant impact of vehicle condition on sale price.

# In[37]:


# 5. Hypothesis Testing - One Way ANOVA
# 6. Data Manipulation
#
-----------------------------------------------------------------------------------------------------------------------

# Hpothesis Test 4: Effect of Fuel Type on Fuel Efficiency:
    # Null Hypothesis: There is no significant difference in fuel efficiency between different fuel types.
    # Alternative Hypothesis: There is a significant difference in fuel efficiency between different fuel types.
    # Test: ANOVA


# In[38]:


# Since my Cars_df already has both EngineType and FuelType I can just use this dataframe for this test. However, I
```

```
# will need to calculate Fuel Efficiency. Using the vehicle's year I will
do so by dividing the mileage for each car by the #
# of years the vehicle has been on the road. Ie; if 2020 vehicle then that
is 4 years, 2021 is 3 years etc.

Cars_df2 = Cars_df # I still want to preserve original Cars_df

current_year = pd.Timestamp.now().year
Cars_df2['Age'] = current_year - Cars_df2['Year']

# Now i have a column for the Age and can calulcate a miles/year to use for
Fuel Efficiency;
Cars_df2['MPY'] = Cars_df2['Mileage'] / Cars_df2['Age']

Cars_df2



# In[39]:



# Create a dictionary to store fuel efficiencies for each fuel type
fuel_efficiencies_by_fuel_type = {}
for fuel_type, group in Cars_df2.groupby('FuelType'):
    fuel_efficiencies_by_fuel_type[fuel_type] = group['MPY']

# Perform ANOVA test
f_statistic, p_value = f_oneway(*fuel_efficiencies_by_fuel_type.values())

# Print results
print("F-statistic:", f_statistic)
print("P-value:", p_value)

# Interpret results
if p_value < 0.05:
    print("Reject null hypothesis: There is a significant effect of engine
type on fuel efficiency.")
else:
    print("Fail to reject null hypothesis: There is no significant effect
of engine type on fuel efficiency.")



# ### Hypothesis Test 4 Interpretation:
#
```

```
#
# F-statistic: The F-statistic is approximately 0.089. This statistic measures the variation in fuel efficiencies between different engine types relative to the variation within each engine type group.
#
#
# P-value: The p-value associated with the F-statistic is approximately 0.966. This p-value represents the probability of observing the data given that the null hypothesis (no significant effect of engine type on fuel efficiency) is true.
#
#
# Overall, since the p-value (0.966) is much greater than the chosen significance level (e.g., 0.05), we fail to reject the null hypothesis. This means that there is insufficient evidence to conclude that there is a significant effect of engine type on fuel efficiency.

# In[40]:


# 5. Hypothesis Testing - Chi-Square and Logistic Regression
# 6. Data Manipulation
#
-----------------------------------------------------------------------------------------------------------------------

# Hpothesis Test 5: Association between Incidents and Market Demand:
    # Null Hypothesis: There is no association between the occurrence of incidents (accidents or damages)
    # and market demand for a car.
    # Alternative Hypothesis: There is an association between the occurrence of incidents and market demand for a car.
    # Test: Chi-square test of independence and logisitc regression


# In[41]:


# First I will sum the # of incidents per carID;

incidents_sum_per_car = Incidents_df.groupby('CarID')['IncidentID'].count().reset_index()
incidents_sum_per_car.rename(columns={'IncidentID': 'TotalIncidents'},
```

```
inplace=True)


# In[42]:


# Now I will add the incidents per carID as a new column at the end of my Cars_df

Cars_df3 = pd.merge(Cars_df, incidents_sum_per_car, on = 'CarID', how = 'left')
Cars_df3


# In[43]:


# If a CarID has NaN as the value for the sum of incidents, that indicates there were 0 incidents and so
# i will replace these NaNs with 0

Cars_df3['TotalIncidents'].fillna(0, inplace=True)
Cars_df3.head(20)


# In[44]:


# Now i am ready to bring over the MarketDemand into my Cars_df3 for testing

data_df3 = pd.merge(Cars_df3, MarketTrends_df[['CarID', 'MarketDemand']], on='CarID', how='left')
data_df3


# In[45]:


# there are some NaN in MarketDemand due to gaps in the MarketTrends with respect to the CarID. I will replace the NaN
# values with the average of the Non-NaN values in MarketDemand
```

```
average_market_demand = data_df3['MarketDemand'].mean()
data_df3['MarketDemand'].fillna(average_market_demand, inplace=True)
data_df3.head(20)


# In[46]:


# Now i can perform my chi-square and logistic regression tests

contingency_table = pd.crosstab(data_df3['TotalIncidents'],
data_df3['MarketDemand'])
chi2, p_value, _, _ = chi2_contingency(contingency_table)

# Print results
print("Chi-square statistic:", chi2)
print("P-value:", p_value)

# Interpret results
if p_value < 0.05:
    print("Reject null hypothesis: There is a significant association
between incidents and market demand.")
else:
    print("Fail to reject null hypothesis: There is no significant
association between incidents and market demand.")

# Perform logistic regression
# Convert 'Incidents' and 'MarketDemand' to binary variables (0 or 1)
data_df3['Incidents_binary'] = np.where(data_df3['TotalIncidents'] > 0, 1,
0)
data_df3['MarketDemand_binary'] = np.where(data_df3['MarketDemand'] > 0, 1,
0)

# Fit logistic regression model
X = data_df3['Incidents_binary']
y = data_df3['MarketDemand_binary']
X = sm.add_constant(X)
logit_model = sm.Logit(y, X)
result = logit_model.fit()

# Print summary of logistic regression model
print(result.summary())
```

```
# ### Hypothesis Test 5 Interpretation:
#
#
# The Chi-square test indicates that there is no significant association
between incidents and market demand.
#
#
# The logistic regression model did not converge due to perfect separation,
making the estimates unreliable.
#
#
# The findings suggest that, based on the available data, there is no
evidence of a significant association between incidents and market demand.

# In[47]:


# 5. Hypothesis Testing - Time-Series Analysis
# 6. Data Manipulation
#
---------------------------------------------------------------------------
-----------------------------------------------

# Hpothesis Test 6: Trend Analysis of Average Sale Price over Time:
    # Null Hypothesis: There is no significant trend in the average sale
price of cars over time.
    # Alternative Hypothesis: There is a significant increasing or
decreasing trend in the average sale price of cars over time.
    # Test: Time-series analysis


# In[48]:


MarketTrends_df2 = MarketTrends_df
MarketTrends_df2


# In[49]:


MarketTrends_df2['Date'] = pd.to_datetime(MarketTrends_df2['Date'])
```

```
MarketTrends_df2


# In[50]:


# 7. Data visualization
#
---------------------------------------------------------------------------
-----------------------------------------------

# My MarketTrends dataframe already has the required columns needed for
this analysis so I will use MarketTrends_df

# Set 'Date' column as index
MarketTrends_df2.set_index('Date', inplace=True)

# Resample the data to monthly frequency and calculate average sale price
for each month
monthly_avg_sale_price =
MarketTrends_df2['AverageSalePrice'].resample('M').mean()

# Plot the time series
plt.figure(figsize=(10, 6))
plt.plot(monthly_avg_sale_price)
plt.title('Average Sale Price Over Time')
plt.xlabel('Date')
plt.ylabel('Average Sale Price')
plt.grid(True)
plt.show()

# Perform time-series decomposition (optional)
decomposition = sm.tsa.seasonal_decompose(monthly_avg_sale_price,
model='additive')
fig = decomposition.plot()
plt.show()


# ### Hypothesis Test 6 Interpretation:
#
#
# Time Series Plot (Top): This shows fluctuations in the average sale price
over time. There's a clear cyclical pattern, suggesting seasonality in the
```

```
data, with peaks and troughs occurring at regular intervals.
#
#
# Time Series Decomposition (Bottom):
#    - Trend: The trend component indicates the long-term progression of
the average sale price. It seems to increase and decrease at different
times, but without a clear overall upward or downward trend over the period
shown.
#
#
#    - Seasonal: The seasonal component captures the regular pattern within
each year, which repeats itself. This could be due to various factors like
market demand changes, sales incentives, or other seasonal factors.
#
#
#    - Residual: The residuals, or the noise in the data, show what's left
after the trend and seasonal components are removed. Ideally, the residuals
should be random and small; however, there are some larger fluctuations,
indicating potential outliers or other patterns not captured by the model.
#
#
#
# This analysis is valuable for understanding the dynamics of sale prices
over time and can help in forecasting future trends or identifying periods
of high or low average sale prices.

# # ---------------------------------- END Python Script
----------------------------------------------
```

1. Link - Data Visualization
2. Full Code:

```
# ## Python Script Overview
#
# This Python code involves data loading, visualization, and analysis tasks
using pandas, matplotlib, seaborn, and wordcloud libraries. Below is a
summary of each section:
#
#    1. Importing Libraries: Imports necessary libraries such as pandas
matplotlib.pyplot, seaborn and wordcloud.
```

```
#
#
#    2. Data Loading:
#        - CSV files containing datasets related to cars, owners, ownership history, vehicle condition, features, incidents, service history, and market trends are loaded into Pandas DataFrames.
#
#
#    3. Data Visualizations:
#        - **Histogram of Car Mileage:** Visualizes the distribution of car mileage using a histogram to understand the range and spread of mileage among the vehicles.
#        - **Bar Chart of Car Makes:** Shows the frequency of different car makes in the dataset using a bar chart.
#        - **Box Plot of Car Prices:** Displays the distribution of car prices using a box plot to identify outliers and understand price ranges.
#        - **Scatter Plot of Car Price vs. Mileage:** Explores the relationship between car price and mileage using a scatter plot.
#        - **Line Chart of Average Sale Price Over Time:** Illustrates the trend of average sale prices of vehicles over time using a line chart.
#        - **Bar Chart of Market Demand by Car Make:** Visualizes the market demand for different car makes using a bar chart.
#        - **Pie Chart of Transmission Types:** Displays the distribution of transmission types among vehicles using a pie chart.
#        - **Heatmap of Correlation Matrix:** Generates a heatmap to visualize the correlation matrix between numerical variables.
#        - **Pair Plot of Select Features:** Creates a pair plot to visualize relationships between multiple variables.
#        - **Violin Plot of Car Prices by Make:** Combines a box plot with a kernel density plot to show the distribution of car prices for each make.
#        - **Bar Chart of Ownership Duration:** Illustrates the frequency of ownership durations for vehicles using a bar chart.
#        - **Line Chart of Mileage Over Time:** Demonstrates how the mileage of vehicles changes over time using a line chart.
#        - **Stacked Bar Chart of Features by Car Make:** Visualizes the prevalence of features for each car make using a stacked bar chart.
#        - **Histogram of Sale Prices by State:** Displays the distribution of sale prices for each state using a histogram.
#        - **Word Cloud of Incident Descriptions:** Creates a word cloud to visualize the most common types of incidents reported for vehicles.
#
#
# Each visualization provides insights into various aspects of the dataset,
```

```
including distributions, trends, correlations, and market demand. These
visualizations aid in understanding the data and extracting valuable
information for analysis and decision-making.
#
# For the code for each of the above components, I will re-iterate with
in-line comments what is mentioned above to identify what code corresponds
to which component from above. Also to keep the context consistent.

# # ----------------------------------- Start Python Script
-----------------------------------------------

# In[1]:


# 1. Importing Necessary Libraries
#
---------------------------------------------------------------------------
------------------------------------------------

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud


# In[2]:


# 2. Data loading - using pandas function ".read_csv()" since in my prior
script I retrieved the data from my SQL database,
# cleaned it, and saved it to .csv files on my own machine.
#
---------------------------------------------------------------------------
------------------------------------------------

Cars_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Cars_df.csv')

Owners_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Owners_df.csv')
```

```
OwnershipHistory_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\OwnershipHistory_df.csv')

VehicleCondition_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\VehicleCondition_df.csv')

Features_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Features_df.csv')

Incidents_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\Incidents_df.csv')

ServiceHistory_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\ServiceHistory_df.csv')

MarketTrends_df = pd.read_csv(r'P:\Users\paulj\Desktop\Important
Documentation\Education\Eastern University\DTSC 691 - Capstone II\Clean
Datasets\MarketTrends_df.csv')


# In[3]:


# 3. Data Visualizations: Histogram of Car Mileage
#
---------------------------------------------------------------------------
------------------------------------------------

# Histogram of Car Mileage: Visualize the distribution of car mileage to
understand the range and spread of
# mileage among the vehicles in your database. This can help identify
common mileage ranges and outliers.

plt.figure(figsize=(10, 6))
plt.hist(Cars_df['Mileage'], bins=20, color='skyblue', edgecolor='black')
# Adjust the number of bins as needed
plt.title('Histogram of Car Mileage')
plt.xlabel('Mileage')
```

```
plt.ylabel('Frequency')
plt.grid(True)
plt.show()


# In[4]:


# 3. Data Visualizations: Bar Chart of Car Makes
# 
--------------------------------------------------------------------------
-----------------------------------------------


# Bar Chart of Car Makes: Create a bar chart showing the frequency of 
different car makes in your database.
# This can provide insights into the most popular car brands among the 
available used vehicles.

car_make_counts = Cars_df['Make'].value_counts()

# Plot bar chart of car makes
plt.figure(figsize=(12, 6))
car_make_counts.plot(kind='bar', color='skyblue')
plt.title('Bar Chart of Car Makes')
plt.xlabel('Car Make')
plt.ylabel('Frequency')
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better 
readability
plt.grid(axis='y')  # Add gridlines to y-axis
plt.tight_layout()  # Adjust layout to prevent overlapping labels
plt.show()


# In[25]:


# 3. Data Visualizations: Box Plot of Car Prices
# 
--------------------------------------------------------------------------
-----------------------------------------------


# Box Plot of Car Prices: Use a box plot to visualize the distribution of 
car prices, including measures of central tendency
```

```
# (median) and variability (interquartile range). This can help identify
outliers and understand the
# price range for different types of vehicles.

# Currently my Cars_df does not have information on the sales prices for
the various vehicles, so I need to
# merge my Cars_df with my OwnershipHistory_df to obtain the sales prices.


data_df = pd.merge(Cars_df, OwnershipHistory_df[['CarID', 'SalePrice']],
on='CarID', how='left')
data_df.head(5)


# In[26]:


# there are some NaN in SalePrice due to gaps in the OwnershipHistory with
respect to the CarID. I will replace the NaN
# values with the average of the Non-NaN values in SalePrice

average_sale_price = data_df['SalePrice'].mean()
data_df['SalePrice'].fillna(average_sale_price, inplace=True)
data_df.head(5)


# In[7]:


# Create a box plot of car prices
plt.figure(figsize=(10, 6))
sns.boxplot(x='SalePrice', data=data_df, color='skyblue')
plt.title('Box Plot of Car Prices')
plt.xlabel('Price')
plt.ylabel('Distribution')
plt.grid(axis='y')  # Add gridlines to y-axis
plt.show()


# In[8]:


# 3. Data Visualizations: Scatter Plot of Car Price vs. Mileage
```

```
#
--------------------------------------------------------------------------------------------------------------------------
# Scatter Plot of Car Price vs. Mileage: Explore the relationship between car price and mileage by creating a scatter plot.
# This can help identify any trends or patterns, such as whether higher mileage correlates with lower prices.

# Create a scatter plot of car price vs. mileage
plt.figure(figsize=(10, 6))
plt.scatter(data_df['Mileage'], data_df['SalePrice'], color='orange', alpha=0.5)
plt.title('Scatter Plot of Car Price vs. Mileage')
plt.xlabel('Mileage')
plt.ylabel('Price')
plt.grid(True)  # Add gridlines
plt.show()


# In[9]:


# 3. Data Visualizations: Line Chart of Average Sale Price Over Time
#
--------------------------------------------------------------------------------------------------------------------------
# Line Chart of Average Sale Price Over Time: If your data includes sale dates, create a line chart showing the
# average sale price of vehicles over time. This can help identify trends in pricing and seasonality effects.

# Group by date and calculate the average sale price
average_price_over_time = MarketTrends_df.groupby('Date')['AverageSalePrice'].mean()

# Convert the index to datetime for proper plotting
average_price_over_time.index = pd.to_datetime(average_price_over_time.index)

# Create a line plot of average sale price over time
plt.figure(figsize=(10, 6))
```

```
average_price_over_time.plot(color='blue', marker='o')
plt.title('Average Sale Price Over Time')
plt.xlabel('Date')
plt.ylabel('Average Sale Price')
plt.grid(True)  # Add gridlines
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.tight_layout()  # Adjust layout to prevent overlapping labels
plt.show()


# The above is quite chaotic with a significant amount of overlap in data
points, making it difficult to discern any clear trends or patterns.
#
# In order to enhance readability and interpretability I will simplofy the
plot by using a 30 day rolling average to smooth out short-term
fluctuations and highlight longer-term trends.

# In[15]:


# 3. Data Visualizations: Line Chart of Average Sale Price Over Time
(Simplified)
#
---------------------------------------------------------------------------
----------------------------------------------

rolling_window_size = 30
average_price_over_time =
MarketTrends_df.groupby('Date')['AverageSalePrice'].mean()
average_price_over_time.index =
pd.to_datetime(average_price_over_time.index)
smoothed_data =
average_price_over_time.rolling(window=rolling_window_size).mean()

plt.figure(figsize=(10, 6))
plt.plot(smoothed_data, color='blue', marker='o', linestyle='-',
linewidth=2, markersize=5)
plt.title('Smoothed Average Sale Price Over Time')
plt.xlabel('Date')
plt.ylabel('Average Sale Price')
plt.grid(True)
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.tight_layout()  # Adjust Layout to prevent overlapping Labels
```

```
plt.show()


# In[10]:


# 3. Data Visualizations: Bar Chart of Market Demand by Car Make
#
---------------------------------------------------------------------------
------------------------------------------------

# Bar Chart of Market Demand by Car Make: If available, visualize the
market demand for different car makes using a bar chart.
# This can provide insights into which car brands are currently in high
demand among buyers.

# Currently my MarketTrends_df does not have information on the make for
the various vehicles, so I need to
# merge my Cars_df with my MarketTrends_df to obtain the vehicle make.

data_df2 = pd.merge(MarketTrends_df, Cars_df[['CarID', 'Make']],
on='CarID', how='left')
data_df2.head()


# In[11]:


# Group by car make and calculate the total market demand
market_demand_by_make = data_df2.groupby('Make')['MarketDemand'].sum()

# Sort the data by market demand in descending order
market_demand_by_make_sorted =
market_demand_by_make.sort_values(ascending=False)

# Create a bar plot of market demand by car make
plt.figure(figsize=(10, 6))
market_demand_by_make_sorted.plot(kind='bar', color='skyblue')
plt.title('Market Demand by Car Make')
plt.xlabel('Car Make')
plt.ylabel('Market Demand')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.grid(axis='y')  # Add gridlines to the y-axis only
```

```
plt.tight_layout()  # Adjust layout to prevent overlapping labels
plt.show()


# The above specific values and rankings are not discernible from the text
description alone. In order to enhance readability and interpretability I
will simplify the plot by grouping my makes into 10 various categories as
follows;
#
# **Luxury Brands:**
#
#    - European Luxury: Aston Martin, Audi, Bentley, BMW, Ferrari,
Lamborghini, Lotus, Maserati, Maybach, McLaren, Mercedes-Benz, Porsche,
Rolls-Royce
#
#    - American Luxury: Cadillac, Lincoln, Tesla
#
#    - Asian Luxury: Acura, Genesis, INFINITI, Lexus
#
#
# **Mainstream Brands:**
#
#    - European Mainstream: Alfa Romeo, FIAT, MINI, Volkswagen, Volvo
#
#    - American Mainstream: Buick, Chevrolet, Chrysler, Dodge, Ford, GMC,
Jeep, Ram
#
#    - Asian Mainstream: Honda, Hyundai, Kia, Mazda, Mitsubishi, Nissan,
Subaru, Toyota
#
#
# **Special Categories:**
#
#    - Exotic/Super Sports: Ferrari, Lamborghini, McLaren
#
#    - Discontinued or Niche: Daewoo, Eagle, Geo, HUMMER, Isuzu, Mercury,
Oldsmobile, Panoz, Plymouth, Pontiac, Saab, Saturn, Scion, smart, Suzuki
#
#
# **Electric Vehicle (EV) and Hybrid Focus:**
#
#    - Dedicated EV Brands: Tesla
#
```

```
#
# **Commercial Vehicles:**
#
#    - Commercial/Fleet: Freightliner

# In[21]:


data_df3 = pd.merge(MarketTrends_df, Cars_df[['CarID', 'Make']],
on='CarID', how='left')

# Dictionary that maps makes to groups
make_to_group = {
    # European Luxury
    'aston martin': 'European Luxury', 'audi': 'European Luxury',
'bentley': 'European Luxury',
    'bmw': 'European Luxury', 'lotus': 'European Luxury', 'maserati':
'European Luxury',
    'maybach': 'European Luxury', 'mercedes-benz': 'European Luxury',
'porsche': 'European Luxury',
    'rolls-royce': 'European Luxury',

    # American Luxury
    'cadillac': 'American Luxury', 'lincoln': 'American Luxury',

    # Asian Luxury
     'acura': 'Asian Luxury', 'genesis': 'Asian Luxury', 'infiniti': 'Asian
Luxury', 'lexus': 'Asian Luxury',

    # European Mainstream
    'alfa romeo': 'European Mainstream', 'fiat': 'European Mainstream',
'mini': 'European Mainstream',
    'volkswagen': 'European Mainstream', 'volvo': 'European Mainstream',

    # American Mainstream
    'buick': 'American Mainstream', 'chevrolet': 'American Mainstream',
'chrysler': 'American Mainstream',
    'dodge': 'American Mainstream', 'ford': 'American Mainstream', 'gmc':
'American Mainstream',
    'jeep': 'American Mainstream', 'ram': 'American Mainstream',

    # Asian Mainstream
    'honda': 'Asian Mainstream', 'hyundai': 'Asian Mainstream', 'kia':
```

```
'Asian Mainstream',
    'mazda': 'Asian Mainstream', 'mitsubishi': 'Asian Mainstream',
'nissan': 'Asian Mainstream',
    'subaru': 'Asian Mainstream', 'toyota': 'Asian Mainstream',

    # Exotic/Super Sports
    'ferrari': 'Exotic/Super Sports', 'lamborghini': 'Exotic/Super Sports',
'mclaren': 'Exotic/Super Sports',

    # Discontinued or Niche
    'daewoo': 'Discontinued or Niche', 'eagle': 'Discontinued or Niche',
'geo': 'Discontinued or Niche',
    'hummer': 'Discontinued or Niche', 'isuzu': 'Discontinued or Niche',
'mercury': 'Discontinued or Niche',
    'oldsmobile': 'Discontinued or Niche', 'panoz': 'Discontinued or
Niche', 'plymouth': 'Discontinued or Niche',
    'pontiac': 'Discontinued or Niche', 'saab': 'Discontinued or Niche',
'saturn': 'Discontinued or Niche',
    'scion': 'Discontinued or Niche', 'smart': 'Discontinued or Niche',
'suzuki': 'Discontinued or Niche',

    # Dedicated EV Brands
    'tesla': 'Dedicated EV',

    # Commercial Vehicles
    'freightliner': 'Commercial/Fleet'
}

# Map the 'Make' column to a new 'Group' column
data_df3['MakeGroup'] = data_df3['Make'].map(make_to_group)

# Fill any missing groups with a default category or leave as NaN
data_df3['MakeGroup'] = data_df3['MakeGroup'].fillna('Other')
data_df3


# In[23]:


# Group by car makegroup and calculate the total market demand
market_demand_by_make = data_df3.groupby('MakeGroup')['MarketDemand'].sum()

# Sort the data by market demand in descending order
```

```
market_demand_by_make_sorted =
market_demand_by_make.sort_values(ascending=False)

# Create a bar plot of market demand by car make
plt.figure(figsize=(10, 6))
market_demand_by_make_sorted.plot(kind='bar', color='skyblue')
plt.title('Market Demand by Car Make Grouping')
plt.xlabel('Make Group')
plt.ylabel('Market Demand')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.grid(axis='y')  # Add gridlines to the y-axis only
plt.tight_layout()  # Adjust layout to prevent overlapping labels
plt.show()


# In[12]:


# 3. Data Visualizations: Pie Chart of Transmission Types
#
---------------------------------------------------------------------------
------------------------------------------------

# Pie Chart of Transmission Types: Create a pie chart to visualize the
distribution of transmission types
# (e.g., automatic, manual) among the vehicles in your database. This can
help understand the prevalence
# of different transmission options.

# Count the frequency of each transmission type
transmission_counts = Cars_df['TransmissionType'].value_counts()

# Plotting
plt.figure(figsize=(8, 8))
plt.pie(transmission_counts, labels=transmission_counts.index,
autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Transmission Types')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a
circle.
plt.show()


# In[13]:
```

```
# 3. Data Visualizations: Heatmap of Correlation Matrix
#
--------------------------------------------------------------------------
-----------------------------------------------

# Heatmap of Correlation Matrix: Generate a heatmap to visualize the
correlation matrix between numerical variables
# such as mileage, price, and year. This can help identify correlations
between different attributes of the vehicles.

# Selecting numerical columns for correlation analysis
numerical_columns = ['Mileage', 'SalePrice', 'Year']  # Adjust as per your
DataFrame

# Calculating correlation matrix
correlation_matrix = data_df[numerical_columns].corr()

# Plotting heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix Heatmap')
plt.show()


# In[14]:


# 3. Data Visualizations: Pair Plot of Select Features
#
--------------------------------------------------------------------------
-----------------------------------------------

# Pair Plot of Select Features: If your database includes additional
features such as engine type or fuel type, create
# a pair plot to visualize relationships between multiple variables
simultaneously. This can help identify interesting
# patterns or clusters in the data.

# Selecting features for pair plot
selected_features = ['Mileage', 'SalePrice', 'Year', 'EngineType',
'FuelType']
```

```
# Creating pair plot
sns.pairplot(data_df[selected_features])
plt.title('Pair Plot of Select Features')
plt.show()


# In[15]:


# 3. Data Visualizations: Violin Plot of Car Prices by Make
#
---------------------------------------------------------------------------
------------------------------------------------

# Violin Plot of Car Prices by Make: This plot combines a box plot with a
kernel density plot to show the distribution
# of car prices for each make. It provides a clearer view of the price
distribution compared to a traditional box plot.

# Set the style of the plot
sns.set(style="whitegrid")

# Create the violin plot
plt.figure(figsize=(12, 6))
sns.violinplot(x='Make', y='SalePrice', data=data_df)
plt.title('Violin Plot of Car Prices by Make')
plt.xlabel('Car Make')
plt.ylabel('SalePrice')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.tight_layout()
plt.show()


# Again, the above specific values and rankings are not discernible from
the text description alone. In order to enhance readability and
interpretability I will use my newly created MakeGroup to simplify the
plot.

# In[28]:


# 3. Data Visualizations: Violin Plot of Car Prices by MakeGroup
```

```
#
--------------------------------------------------------------------------------------------------------------------------------------------
# Dictionary that maps makes to groups
make_to_group = {
    # European Luxury
    'aston martin': 'European Luxury', 'audi': 'European Luxury',
'bentley': 'European Luxury',
    'bmw': 'European Luxury', 'lotus': 'European Luxury', 'maserati':
'European Luxury',
    'maybach': 'European Luxury', 'mercedes-benz': 'European Luxury',
'porsche': 'European Luxury',
    'rolls-royce': 'European Luxury',

    # American Luxury
    'cadillac': 'American Luxury', 'lincoln': 'American Luxury',

    # Asian Luxury
     'acura': 'Asian Luxury', 'genesis': 'Asian Luxury', 'infiniti': 'Asian
Luxury', 'lexus': 'Asian Luxury',

    # European Mainstream
    'alfa romeo': 'European Mainstream', 'fiat': 'European Mainstream',
'mini': 'European Mainstream',
    'volkswagen': 'European Mainstream', 'volvo': 'European Mainstream',

    # American Mainstream
    'buick': 'American Mainstream', 'chevrolet': 'American Mainstream',
'chrysler': 'American Mainstream',
    'dodge': 'American Mainstream', 'ford': 'American Mainstream', 'gmc':
'American Mainstream',
    'jeep': 'American Mainstream', 'ram': 'American Mainstream',

    # Asian Mainstream
    'honda': 'Asian Mainstream', 'hyundai': 'Asian Mainstream', 'kia':
'Asian Mainstream',
    'mazda': 'Asian Mainstream', 'mitsubishi': 'Asian Mainstream',
'nissan': 'Asian Mainstream',
    'subaru': 'Asian Mainstream', 'toyota': 'Asian Mainstream',

    # Exotic/Super Sports
    'ferrari': 'Exotic/Super Sports', 'lamborghini': 'Exotic/Super Sports',
```

```
'mclaren': 'Exotic/Super Sports',

    # Discontinued or Niche
    'daewoo': 'Discontinued or Niche', 'eagle': 'Discontinued or Niche',
'geo': 'Discontinued or Niche',
    'hummer': 'Discontinued or Niche', 'isuzu': 'Discontinued or Niche',
'mercury': 'Discontinued or Niche',
    'oldsmobile': 'Discontinued or Niche', 'panoz': 'Discontinued or
Niche', 'plymouth': 'Discontinued or Niche',
    'pontiac': 'Discontinued or Niche', 'saab': 'Discontinued or Niche',
'saturn': 'Discontinued or Niche',
    'scion': 'Discontinued or Niche', 'smart': 'Discontinued or Niche',
'suzuki': 'Discontinued or Niche',

    # Dedicated EV Brands
    'tesla': 'Dedicated EV',

    # Commercial Vehicles
    'freightliner': 'Commercial/Fleet'
}

# Map the 'Make' column to a new 'Group' column
data_df['MakeGroup'] = data_df['Make'].map(make_to_group)

# Fill any missing groups with a default category or leave as NaN
data_df['MakeGroup'] = data_df['MakeGroup'].fillna('Other')

# Set the style of the plot
sns.set(style="whitegrid")

# Create the violin plot
plt.figure(figsize=(12, 6))
sns.violinplot(x='MakeGroup', y='SalePrice', data=data_df)
plt.title('Violin Plot of Car Prices by Make Grouping')
plt.xlabel('Make Group')
plt.ylabel('SalePrice')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.tight_layout()
plt.show()



# In[29]:

```

```
# 3. Data Visualizations: Bar Chart of Ownership Duration
#
------------------------------------------------------------------------------------------------------------------------
# Bar Chart of Ownership Duration: Calculate the duration of ownership for each vehicle (SaleDate - PurchaseDate)
# and create a bar chart showing the frequency of ownership durations. This can provide insights into how long
# owners typically keep their vehicles before selling them.

# Convert 'PurchaseDate' and 'SaleDate' columns to datetime objects
OwnershipHistory_df['PurchaseDate'] = pd.to_datetime(OwnershipHistory_df['PurchaseDate'])
OwnershipHistory_df['SaleDate'] = pd.to_datetime(OwnershipHistory_df['SaleDate'])

# Calculate ownership duration (in days) for each vehicle
OwnershipHistory_df['OwnershipDuration'] = (OwnershipHistory_df['SaleDate'] - OwnershipHistory_df['PurchaseDate']).dt.days

# Create a bar chart of ownership durations
plt.figure(figsize=(10, 6))
OwnershipHistory_df['OwnershipDuration'].value_counts().sort_index().plot(kind='bar', color='skyblue')
plt.title('Bar Chart of Ownership Duration')
plt.xlabel('Ownership Duration (Days)')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()


# Due to the compressed scale and the volume of data, it's challenging to discern specific patterns or to identify the most common ownership duration from this visualization. In order to enhance readability and interpretability I will simplify the plot by adjusting the bin sizes.

# In[30]:


ownership_durations = OwnershipHistory_df['OwnershipDuration']
```

```
# Define the number of bins or the specific bin edges you want
# For example, to use bin sizes of 30 days (approximately 1 month), you can
calculate the bin range like this:
bin_size = 30  # days
max_duration = ownership_durations.max()
bins = range(0, max_duration + bin_size, bin_size)

# Create a histogram with the defined bins
plt.figure(figsize=(10, 6))
plt.hist(ownership_durations, bins=bins, color='skyblue',
edgecolor='black')
plt.title('Histogram of Ownership Duration')
plt.xlabel('Ownership Duration (Days)')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()


# In[17]:


# 3. Data Visualizations: Line Chart of Mileage Over Time
#
---------------------------------------------------------------------------
------------------------------------------------

# Line Chart of Mileage Over Time: If your data includes mileage readings
over time (e.g., from service history records),
# create a line chart showing how the mileage of vehicles changes over
time. This can help identify trends in
# mileage accumulation and potential patterns related to vehicle usage.

# Currently my ServiceHistory_df does not have information on the mileage
for the various vehicles, so I need to
# merge my ServiceHistory_df with my Cars_df to obtain the mileage.

data_df3 = pd.merge(ServiceHistory_df, Cars_df[['CarID', 'Mileage']],
on='CarID', how='left')
data_df3.head()
```

```
# In[18]:


# Convert 'ServiceDate' column to datetime object
data_df3['ServiceDate'] = pd.to_datetime(data_df3['ServiceDate'])

# Group by 'ServiceDate' and calculate the average mileage for each date
mileage_over_time = data_df3.groupby('ServiceDate')['Mileage'].mean()

# Create a line chart of mileage over time
plt.figure(figsize=(10, 6))
mileage_over_time.plot(kind='line', color='green', marker='o',
linestyle='-')
plt.title('Line Chart of Mileage Over Time')
plt.xlabel('Service Date')
plt.ylabel('Average Mileage')
plt.grid(True)
plt.tight_layout()
plt.show()


# In[19]:


# 3. Data Visualizations: Stacked Bar Chart of Features by Car Make
#
---------------------------------------------------------------------------
------------------------------------------------

# Stacked Bar Chart of Features by Car Make: If your data includes features
such as air conditioning, power windows,
# etc., create a stacked bar chart showing the prevalence of these features
for each car make.
# This can help identify which features are most common for different
brands.

# Pivot the DataFrame to get a count of each feature by car make
feature_counts = Features_df.pivot_table(index='CarID',
columns='FeatureName', aggfunc='size', fill_value=0)

# Merge with the Cars DataFrame to get the make of each car
feature_counts = feature_counts.merge(Cars_df[['CarID', 'Make']],
```

```
on='CarID', how='left')

# Group by make and sum the counts of each feature
feature_counts_by_make = feature_counts.groupby('Make').sum()

# Plot a stacked bar chart
plt.figure(figsize=(12, 8))
feature_counts_by_make.plot(kind='bar', stacked=True)
plt.title('Stacked Bar Chart of Features by Car Make')
plt.xlabel('Car Make')
plt.ylabel('Count')
plt.xticks(rotation=45, ha='right')
plt.legend(title='Feature Name', bbox_to_anchor=(1.05, 1), loc='upper
left')
plt.tight_layout()
plt.show()


# The above specific values and rankings are not discernible from the text
description alone. In order to enhance readability and interpretability I
will simplify the plot by using my MakeGroup column.

# In[31]:


# 3. Data Visualizations: Stacked Bar Chart of Features by Car Make Group
#
---------------------------------------------------------------------------
-----------------------------------------------

data_df4 = Cars_df

# Dictionary that maps makes to groups
make_to_group = {
    # European Luxury
    'aston martin': 'European Luxury', 'audi': 'European Luxury',
'bentley': 'European Luxury',
    'bmw': 'European Luxury', 'lotus': 'European Luxury', 'maserati':
'European Luxury',
    'maybach': 'European Luxury', 'mercedes-benz': 'European Luxury',
'porsche': 'European Luxury',
    'rolls-royce': 'European Luxury',
```

```
    # American Luxury
    'cadillac': 'American Luxury', 'lincoln': 'American Luxury',

    # Asian Luxury
     'acura': 'Asian Luxury', 'genesis': 'Asian Luxury', 'infiniti': 'Asian
Luxury', 'lexus': 'Asian Luxury',

    # European Mainstream
    'alfa romeo': 'European Mainstream', 'fiat': 'European Mainstream',
'mini': 'European Mainstream',
    'volkswagen': 'European Mainstream', 'volvo': 'European Mainstream',

    # American Mainstream
    'buick': 'American Mainstream', 'chevrolet': 'American Mainstream',
'chrysler': 'American Mainstream',
    'dodge': 'American Mainstream', 'ford': 'American Mainstream', 'gmc':
'American Mainstream',
    'jeep': 'American Mainstream', 'ram': 'American Mainstream',

    # Asian Mainstream
    'honda': 'Asian Mainstream', 'hyundai': 'Asian Mainstream', 'kia':
'Asian Mainstream',
    'mazda': 'Asian Mainstream', 'mitsubishi': 'Asian Mainstream',
'nissan': 'Asian Mainstream',
    'subaru': 'Asian Mainstream', 'toyota': 'Asian Mainstream',

    # Exotic/Super Sports
    'ferrari': 'Exotic/Super Sports', 'lamborghini': 'Exotic/Super Sports',
'mclaren': 'Exotic/Super Sports',

    # Discontinued or Niche
    'daewoo': 'Discontinued or Niche', 'eagle': 'Discontinued or Niche',
'geo': 'Discontinued or Niche',
    'hummer': 'Discontinued or Niche', 'isuzu': 'Discontinued or Niche',
'mercury': 'Discontinued or Niche',
    'oldsmobile': 'Discontinued or Niche', 'panoz': 'Discontinued or
Niche', 'plymouth': 'Discontinued or Niche',
    'pontiac': 'Discontinued or Niche', 'saab': 'Discontinued or Niche',
'saturn': 'Discontinued or Niche',
    'scion': 'Discontinued or Niche', 'smart': 'Discontinued or Niche',
'suzuki': 'Discontinued or Niche',

    # Dedicated EV Brands
```

```
    'tesla': 'Dedicated EV',

    # Commercial Vehicles
    'freightliner': 'Commercial/Fleet'
}

# Map the 'Make' column to a new 'Group' column
data_df4['MakeGroup'] = data_df4['Make'].map(make_to_group)

# Fill any missing groups with a default category or leave as NaN
data_df['MakeGroup'] = data_df['MakeGroup'].fillna('Other')


# Pivot the DataFrame to get a count of each feature by car make
feature_counts = Features_df.pivot_table(index='CarID',
columns='FeatureName', aggfunc='size', fill_value=0)

# Merge with the Cars DataFrame to get the make of each car
feature_counts = feature_counts.merge(data_df4[['CarID', 'MakeGroup']],
on='CarID', how='left')

# Group by make and sum the counts of each feature
feature_counts_by_make = feature_counts.groupby('MakeGroup').sum()

# Plot a stacked bar chart
plt.figure(figsize=(12, 8))
feature_counts_by_make.plot(kind='bar', stacked=True)
plt.title('Stacked Bar Chart of Features by Car Make Grouping')
plt.xlabel('Make Group')
plt.ylabel('Count')
plt.xticks(rotation=45, ha='right')
plt.legend(title='Feature Name', bbox_to_anchor=(1.05, 1), loc='upper
left')
plt.tight_layout()
plt.show()


# In[33]:


# 3. Data Visualizations: Histogram of Sale Prices by State
#
---------------------------------------------------------------------------
```

```
-----------------------------------------------

# Histogram of Sale Prices by State: If your data includes the state where
each sale occurred, create a histogram
#showing the distribution of sale prices for each state. This can help
identify regional differences in pricing
# and market conditions.

# Currently my OwnershipHistory_df does not have information on the state
for the various vehicles, so I need to
# merge my OwnershipHistory_df with my Owners_df to obtain the mileage.

data_df5 = pd.merge(OwnershipHistory_df, Owners_df[['OwnerID', 'State']],
on='OwnerID', how='left')
data_df5.head()


# In[21]:


# Filter out any missing or invalid sale prices
valid_sale_prices = data_df5['SalePrice'].dropna()

# Plot a histogram of sale prices for each state
plt.figure(figsize=(12, 8))
for state in data_df5['State'].unique():
    state_sale_prices = data_df5.loc[data_df5['State'] == state,
'SalePrice']
    plt.hist(state_sale_prices, bins=20, alpha=0.5, label=state,
density=True)

plt.title('Histogram of Sale Prices by State')
plt.xlabel('Sale Price')
plt.ylabel('Frequency')
plt.legend(title='State', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.tight_layout()
plt.show()


# Due to the compressed scale and the volume of data, it's challenging to
discern specific patterns from this visualization. In order to enhance
readability and interpretability I will simplify the plot by grouping the
```

```
states based on geogrpahic location as follows;
#
# **Northeast:**
#
#    - New England: CT, MA, ME, NH, RI, VT
#
#    - Mid-Atlantic: NJ, NY, PA
#
#
# **Midwest:**
#
#    - East North Central: IL, IN, MI, OH, WI
#
#    - West North Central: IA, KS, MN, MO, ND, NE, SD
#
#
# **South:**
#
#    - South Atlantic: DC, DE, FL, GA, MD, NC, SC, VA, WV
#
#    - East South Central: AL, KY, MS, TN
#
#    - West South Central: AR, LA, OK, TX
#
#
# **West:**
#
#    - Mountain: AZ, CO, ID, MT, NM, NV, UT, WY
#    - Pacific: AK, CA, HI, OR, WA
#
#
# **Territories:**
#
#    - U.S. Territories: AS (American Samoa), FM (Federated States of
Micronesia), GU (Guam), MH (Marshall Islands), MP (Northern Mariana
Islands), PR (Puerto Rico), PW (Palau), VI (U.S. Virgin Islands)

# In[35]:


# Define a dictionary mapping states to location categories
state_to_location = {
    # Northeast
```

```
    'CT': 'New England', 'MA': 'New England', 'ME': 'New England', 'NH':
'New England',
    'RI': 'New England', 'VT': 'New England', 'NJ': 'Mid-Atlantic', 'NY':
'Mid-Atlantic',
    'PA': 'Mid-Atlantic',

    # Midwest
    'IL': 'East North Central', 'IN': 'East North Central', 'MI': 'East
North Central',
    'OH': 'East North Central', 'WI': 'East North Central', 'IA': 'West
North Central',
    'KS': 'West North Central', 'MN': 'West North Central', 'MO': 'West
North Central',
    'ND': 'West North Central', 'NE': 'West North Central', 'SD': 'West
North Central',

    # South
    'DC': 'South Atlantic', 'DE': 'South Atlantic', 'FL': 'South Atlantic',
'GA': 'South Atlantic',
    'MD': 'South Atlantic', 'NC': 'South Atlantic', 'SC': 'South Atlantic',
'VA': 'South Atlantic',
    'WV': 'South Atlantic', 'AL': 'East South Central', 'KY': 'East South
Central',
    'MS': 'East South Central', 'TN': 'East South Central', 'AR': 'West
South Central',
    'LA': 'West South Central', 'OK': 'West South Central', 'TX': 'West
South Central',

    # West
    'AZ': 'Mountain', 'CO': 'Mountain', 'ID': 'Mountain', 'MT': 'Mountain',
'NM': 'Mountain',
    'NV': 'Mountain', 'UT': 'Mountain', 'WY': 'Mountain', 'AK': 'Pacific',
'CA': 'Pacific',
    'HI': 'Pacific', 'OR': 'Pacific', 'WA': 'Pacific',

    # Territories
    'AS': 'U.S. Territories', 'FM': 'U.S. Territories', 'GU': 'U.S.
Territories',
    'MH': 'U.S. Territories', 'MP': 'U.S. Territories', 'PR': 'U.S.
Territories',
    'PW': 'U.S. Territories', 'VI': 'U.S. Territories'
}
```

```python
data_df5['Location'] = data_df5['State'].map(state_to_location)

# Plot a histogram of sale prices for each location
plt.figure(figsize=(12, 8))
for location in data_df5['Location'].unique():
    state_sale_prices = data_df5.loc[data_df5['Location'] == location,
'SalePrice']
    plt.hist(state_sale_prices, bins=20, alpha=0.5, label=location,
density=True)

plt.title('Histogram of Sale Prices by Geographic Location')
plt.xlabel('Sale Price')
plt.ylabel('Frequency')
plt.legend(title='Location', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.tight_layout()
plt.show()


# Still not as clear - let's try as a boxplot.

# In[37]:


sale_prices_by_location = [data_df5[data_df5['Location'] ==
location]['SalePrice'].values for location in
data_df5['Location'].unique()]
plt.figure(figsize=(12, 8))
plt.boxplot(sale_prices_by_location, labels=data_df5['Location'].unique())
plt.title('Boxplot of Sale Prices by Geographic Location')
plt.xlabel('Location')
plt.ylabel('Sale Price')
plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()
plt.show()


# In[22]:


# 3. Data Visualizations: Word Cloud of Incident Descriptions
#
```

```
--------------------------------------------------------------------------------------------------------------------------

# Word Cloud of Incident Descriptions: If your data includes incident descriptions, create a word cloud to visualize
# the most common types of incidents reported for the vehicles in your database. This can provide insights into common
#issues or concerns with different types of vehicles.

# Combine all incident descriptions into a single string and convert to lowercase
all_descriptions = ' '.join(Incidents_df['Description'].dropna()).lower()

# Generate the word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_descriptions)

# Plot the word cloud
plt.figure(figsize=(10, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.title('Word Cloud of Incident Descriptions')
plt.axis('off')
plt.show()


# # ----------------------------------- END Python Script ----------------------------------------------
```

C. DBML (For ER Diagram)

```
Table Cars {
  CarID Int [PK]
  Make Varchar
  Model Varchar
  Year Int
  Mileage Int
  VIN Varchar
  EngineType Varchar
  TransmissionType Varchar
  FuelType Varchar
}
```

```
Table Owners {
  OwnerID Int [PK]
  CarID Int [ref: > Cars.CarID]
  FirstName Varchar
  LastName Varchar
  ContactInfo Varchar
  State Varchar
}

Table OwnershipHistory {
  OwnershipID Int [PK]
  CarID Int [ref: > Cars.CarID]
  OwnerID Int [ref: > Owners.OwnerID]
  PurchaseDate Date
  SaleDate Date
  SalePrice Decimal
}

Table VehicleCondition {
  ConditionID Int [PK]
  CarID Int [ref: > Cars.CarID]
  OverallCondition Varchar
  ExteriorCondition Varchar
  InteriorCondition Varchar
}

Table Features {
  FeatureID Int [PK]
  CarID Int [ref: > Cars.CarID]
  FeatureName Varchar
  FeatureValue Varchar
}

Table Incidents {
  IncidentID Int [PK]
  CarID Int [ref: > Cars.CarID]
  IncidentDate Date
  Description Text
```

```
  SalePrice Decimal
}


Table ServiceHistory {
  ServiceID Int [PK]
  CarID Int [ref: > Cars.CarID]
  ServiceDate Date
  ServiceType Varchar
  Cost Decimal
}

Table MarketTrends {
  TrendID Int [PK]
  CarID Int [ref: > Cars.CarID]
  Date Date
  AverageSalePrice Decimal
  MarketDemand Int
}
```

D. Presentation Slideshow