CPSC 437/537 Project: CarMin

Ruomu (Felix) Zou and Emily Goldfarb

Code:

GitHub Repository Link: https://github.com/zouruomu/CarMin

Introduction:

The process of purchasing a used car is notorious for being stressful and full of individuals or dealerships eager to exploit the buyer's lack of information. The wide price disparity between different car models, varying mileage, service/accident histories, and lack of readily available information commonly lead uninformed buyers to pay thousands more than market price only for their "new" car to break down soon after the purchase. Our goal was to create a database with modeling capacities to help the user find a good used car stress-free.

Our database aims to support:

- The ability to efficiently store a large amount of data spanning all vehicle types, dealer regions, and production/listing dates so that users have more knowledge to tap into.
- The capability for the user to make arbitrary SQL queries so they can look up anything they want.
- The capability for the user to easily enter some details (such as make, model, year, mileage, etc.) about a category of vehicle they are looking for and get the predicted price given the criteria entered.

Basic Architecture:

Our project is organized into 5 major sections (each of which is elaborated upon below):

- Section 1: Database Creation.
- Section 2: Loading Database into MySQL.
- Section 3: Train Model to Predict Used Car Prices.
- Section 4: Interactive Jupyter Notebook Demo.
- Section 5: Plotting and EDA using R.

Description of data sources:

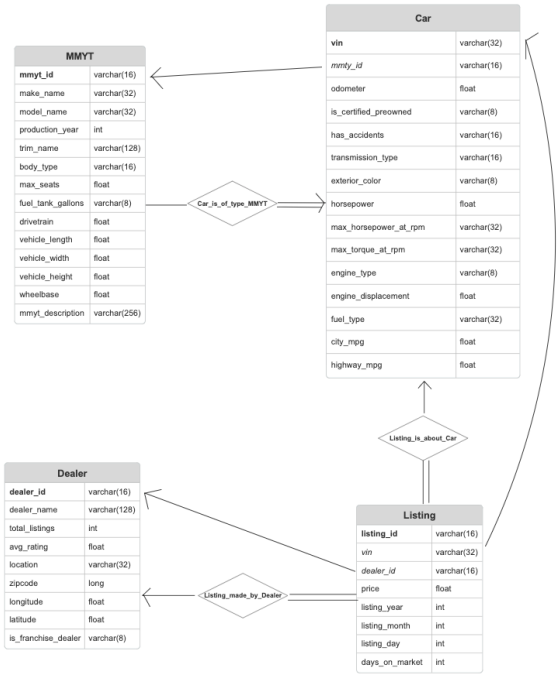
- US Used Cars Dataset:
  - Source: Kaggle.
  - Link: https://www.kaggle.com/datasets/ananaymital/us-used-cars-dataset?rvi=1.
  - Approximately 3 million entries.
  - Data was collected from Car Gurus website inventory.
- Used Cars Dataset:
  - Source: Kaggle.
  - Link: https://www.kaggle.com/datasets/austinreese/craigslist-carstrucks-data.
  - Approximately 460,000 entries.
  - Data was collected from Craigslist.

Database schema:

Our two datasets were cleaned in Python, loaded into MySQL and then joined through a set of queries. The schema of our database is as follows:

- Car(**vin**, *mmty_id*, odometer, is_certified_preowned, has_accidents, transmission_type, exterior_color, horsepower, max_horsepower_at_rpm, max_torque_at_rpm, engine_type, engine_displacement, fuel_type, city_mpg, highway_mpg)
- MMYT(**mmyt_id**, make_name, model_name, production_year, trim_name, body_type, max_seats, fuel_tank_gallons, drivetrain, vehicle_length, vehicle_width, vehicle_height, wheelbase, mmyt_description)
- Dealer(**dealer_id**, dealer_name, total_listings, avg_rating, location, zipcode, longitude, latitude, is_franchise_dealer)
- Listing(**listing_id**, *vin*, *dealer_id*, price, listing_year, listing_month, listing_day, days_on_market)

Primary keys are bolded and foreign keys are italicized. Car details an instance of a used car, MMYT notably holds the make, model, year, and trim of a car, Dealer has information about the sellers of these used cars, and Listing details the advertisement information. There is a many to one relationship between Car and MMYT because MMYT can be associated with multiple Car instances while each Car instance is associated with one MMYT instance. Car has total participation with MMYT and the relationship between Car and MMYT is labeled Car_is_of_type_MMYT. There is also a many to one relationship from Listing to Car; cars have the potential to be sold multiple times and thus appear in Listing multiple times (this does in fact happen in our dataset). However, Listing has total participation with Car and is associated with only one instance of Car. The relationship between Listing and Car is labeled as Listing_is_about_Car. Our third and final relationship, Listing_made_by_Dealer, is a many to one relationship from Listing to Dealer. Dealers can list multiple cars but each Listing instance is made by exactly one Dealer. There is total participation from Listing in this relationship. Below is our ER diagram:



Description and Key Features of Section 1: Database Creation

Relevant code in repository:

- Section1_DatabaseCreation.ipynb
- Section1_DatabaseCreation.html

Note: The .html file is a fully rendered version of the Jupyter Notebook and contains the exact same code. This is done to allow graders to easily view the work completed. If desired, the Jupyter Notebook can be run to verify results, etc.

Extra credit attempted: See the “note to grader” at the very beginning of the file, which identifies **Part 1.4**, **Part 3.3**, and **Part 3.4** of the notebook as attempting extra credit.

Basic overview:

Our two datasets combined totals almost 3,500,000 used cars. Both datasets came from Kaggle, but the original data was pulled from Car Gurus (a used car website) and Craigslist. The first dataset contains 3,000,000 entries and over 60 columns. The second set is still large but is considerably smaller than the first dataset with 426,000 observations across 26 columns. Handling entity resolution proved to be an undertaking with our vast amount of data. It is important to note that the same car was allowed to appear multiple times across the datasets because it was feasible that the same car could be sold multiple times.

Technical challenge 1: Large data volume

The first major challenge is efficient representation of large volumes of raw data with a lot of redundancy. To better put the concepts of normalization and good relational database design learned in class into practice, we decided to aggressively decompose the ~10GB of data to remove redundancy. The final database is less than 1GB and actually fits completely into a GitHub repository!

Of course, such aggressive decomposition would result in greater query times as more tables need to be joined for each query. However, we viewed this as a worthwhile tradeoff. Furthermore, we also took great care to make sure that our decompositions are lossless:

-

This decomposition should be lossless, is it? As a final check, let's make extra sure that, if we merge the splitted relations, we get the original:

```
merged = listings.merge(
            cars.merge(mmyts, on="trimId"),on="vin").merge(
                dealers, on="sp_id")[Data1_raw.columns].sort_values("vin").reset_index(drop=True)
orig_sorted = Data_raw.drop_duplicates().sort_values("vin").reset_index(drop=True)
(merged == orig_sorted).all().all()
```

```
True
```

Yep, exactly the same!

Technical challenge 2: (Very) messy data

All raw data is messy, ours especially so. Here are a couple examples of a simple form of messiness that is easily detected and fixed (for details see the notebook):

`engine_cylinders` has the exact same entries as `engine_type`:

```
(Data1_raw.engine_cylinders.dropna() == Data1_raw.engine_type.dropna()).all()
```

```
True
```

`county` has only null values:

```
Data2_raw.county.isnull().all()
```

```
True
```

If only everything was this simple! There are more elusive sources of messiness too:

As stated before, `VIN` is highly important and should always be 17 digits, is that the case in this dataset?

```
Data2_raw[Data2_raw.VIN.str.len() != 17].VIN
```

```
338       11402312009097
1016          0906419252
1684       116356W169501
1698          7275856000
2524          B1DL120963
                 ...
425059      31847S147227
425300     1Z37L6S418690
426339       CR315045444
426373     CCL449J162701
426376     CCL449F505274
Name: VIN, Length: 1318, dtype: object
```

Nope... don't know what's going on with these VIN's that don't any sense. There are even a couple `VIN` entries with only 1 digit:

```
Data2_raw[Data2_raw.VIN.str.len() == 1].VIN
```

```
25010    C
27561    C
Name: VIN, dtype: object
```

So far, all these can easily be detected if one looks carefully. But about values that are consistent with the database but wrong in reality?

```
Data1_raw.loc[Data1_raw.trimId == "MMYT047624", ["trimId", "make_name", "model_name", "trim_name", "year", "body_typ
```

	trimId	make_name	model_name	trim_name	year	body_type
129215	MMYT047624	Toyota	Prius Plug-In	Base	2012	Sedan
149803	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
172121	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
196247	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
239168	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
1434532	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
1694471	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
2256008	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
2423967	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback
2529964	MMYT047624	Toyota	Prius Plug-In	Base	2012	Hatchback

In this example, the body_type of a Toyota Prius was mistakenly listed by a dealer as “Sedan” when it should be a “hatchback.” This prevents some of our generated ID columns from uniquely identifying tuples and therefore must be dealt with on a case-by-case basis.

There were many more uncleannesses just like these throughout the data! Unsurprisingly, cleaning this data took *a lot* of work, which is also why the relevant notebook was so long. For details on what exact measures were taken to correct each problem, refer to **Parts 1 and 3** of the notebook for details.

Technical challenge 3: Entity resolution

Since our data came from such different sources and are scraped from completely different websites, it is not surprising that the entities are formatted differently and sometimes contain slightly different information. See below for a couple examples of what were done to resolve entities.

Matching differently formatted values:

```
Data2_raw["type"] = Data2_raw.type.replace({"sedan":"Sedan",
                                            "SUV":"SUV / Crossover",
                                            "pickup":"Pickup Truck",
                                            "truck":"Pickup Truck",
                                            "other":"Other",
                                            "coupe":"Coupe",
                                            "hatchback":"Hatchback",
                                            "wagon":"Wagon",
                                            "van":"Van",
                                            "convertible":"Convertible",
                                            "mini-van":"Minivan",
                                            "offroad":"SUV / Crossover",
                                            "bus":"Pickup Truck"})
Data2_raw["type"] = Data2_raw.type.fillna("Unknown")
```

Generating ID columns:

```
Data1_mmyts.mmyt_id.str[4:].astype(int).describe()
```

```
count    36964.000000
mean     49635.126231
std      25138.045590
min          3.000000
25%      32179.750000
50%      48896.000000
75%      72480.250000
max      94257.000000
Name: mmyt_id, dtype: float64
```

Looks like we are safe to use anything beyond `"MMYT094257"` ! Let's start from `"MMYT095000"` to have some separation in the ID space and go sequentially from there:

```
Data2_mmyts.reset_index(drop=True, inplace=True)
Data2_mmyts["mmyt_id"] = "MMYT" + pd.Series(range(95000, 95000 + Data2_mmyts.shape[0])).astype(str).str.zfill(6)
```

… And much more! See **Parts 2 and 4** of the notebook for details.

End result of Section 1:

The end result of section is a database consisting of the following files, each corresponding to a relation in the schema:

- MMYT.csv
- Car.csv
- Dealer.csv
- Listing.csv

Onto loading these files into MySQL!

Description and Key Features of Section 2: Loading Database into MySQL.

Relevant code in repository:

- Section2_LoadIntoMySQL.sql

Basic overview:

This section is straightforward and uses a .sql script to create the database, create tables, and load data from files in. A nuance here is that the global variable `SET GLOBAL local_infile=1;` has to be set, and the script must be run with the option `--local-infile=1`. All in all, everything here can be run with just the command:

mysql -u root -p --local-infile=1 < ./Section2_LoadIntoMySQL.sql

End result of Section 2:

By the end of this section, a MySQL database named CarMin should have been created on the user's local machine. It follows all typical MySQL formats and can be accessed from terminal directly or, as we recommend, through the code in Sections 3 and 4.

Description and Key Features of Section 3: Train Model to Predict Used Car Prices.

Relevant code in repository:

- Section3_TrainModel.ipynb
- Section3_TrainModel.html

Note: The .html file is a fully rendered version of the Jupyter Notebook and contains the exact same code. This is done to allow graders to easily view the work completed. If desired, the Jupyter Notebook can be run to verify results, etc.

Extra credit attempted: We hope that this entire section may be assessed under the "extraordinary extra features" section of the rubric for extra credit if deemed deserving!

Basic overview:

Having a database is nice, but it is no use if insights cannot be derived from it. Often, with such large volumes of data, machine algorithms catch on to patterns much better than human observers. Therefore, we decided to train a model to take in aspects of a car a user might be considering, such as its make, model, color, odometer reading, etc., and provide the user with an estimate of how much they should expect to pay. The model querying is done in Section 4, and Section 3 (this one) is responsible for creating and training the model.

During our exploratory analysis in Section 1 (specifically, see **Section 1 Part 3.4**), we have deemed discrete models to be suitable in this context. A simple example, drawn from Part 3.4, to motivate this decision is shown here:

```
camaros = Data2_raw[Data2_raw.model.str.contains("camaro", na=False)]
```

What's the price like?

```
camaros.price.describe()
```

```
count      1488.000000
mean      28000.123656
std       12197.341473
min           1.000000
25%       18995.000000
50%       27989.000000
75%       37990.000000
max      129991.000000
Name: price, dtype: float64
```

Average seems a little low.. what if we split by 5th gen and 6th gen?

```
camaros[camaros.year <= 2015].price.mean()
```

```
21272.29655172414
```

```
camaros[camaros.year > 2015].price.mean()
```

```
34392.882044560945
```

Makes much more sense -> this a good example of why more discreet methods (trees?) actually might work better for this problem.

As can be seen, an increase by 1 on a continuous variable (production_year) can cause massive disparities in price due to the car changing generations in that year. Continuous methods would need more feature engineering to be able to catch onto all of these nuances. In the end, we settled on using an LGBM – which is an ensemble of gradient boosted trees that are built leaf-wise – for its balance of speed and performance. Below is our model specification and training:

```
lgb_params = {
    "objective":"regression",
    "boosting":"gbdt",
    "learning_rate":0.1,
    "num_leaves":1024, # there are a lot of data and problem is regression
    "max_depth":-1, # lgb grows trees sideways
    "min_data_in_leaf":80,
    "metric":"l1",
    "seed":2830,
    "verbose": -1
}
num_iterations = 1000
lgb_model = lgb.train(lgb_params, dataset_train, num_iterations,
                      valid_sets=[dataset_validate], verbose_eval=False)
lgb_model.save_model("./Models/MODEL_LGBM.txt")
```

The model tapped into the database using the following complex SQL query:

```
# connect to MySQL Server -> assumes Section2_LoadIntoMySQL.sql has already been run
# NOTE: For demo purposes, at present, the password for MySQL server on my local machine
#       is temporarily set to "insecure_password"
mysql_connection = pymysql.connect(host="localhost", user="root", password = "insecure_password", db="CarMin")
cursor = mysql_connection.cursor()

# complex query to decompress compressed data
sql_get_full_table = """
                     SELECT *
                     FROM Listing
                         LEFT OUTER JOIN (Car LEFT OUTER JOIN MMYT USING (mmyt_id)) USING (vin)
                         LEFT OUTER JOIN Dealer USING (dealer_id);
                     """

# run query through MySQL and store result in pandas dataframe
data_all = pd.read_sql_query(sql_get_full_table, mysql_connection)

# finally, close connection
mysql_connection.close()
```

Furthermore, the data was split as follows:

- Training set: 80% of data.
- Validation set: 10% of data.
- Testing set: 10% of data.

Note: We did not opt for cross validation as we have a lot of data and any fold CV would take too long.

We did some minimal hyperparameter tuning (mostly num_leaves, learning_rate, metric, and num_iterations) using the validation set, but were careful not to over-tune lest we overfit to the validation set. On the whole, training and testing losses indicate that our model has not overfitted and has thoroughly converged by the end of the 1000 iterations. Here is the final testing error:

```
test_predictions = lgb_model.predict(dataset_test.data)
mean_absolute_error(dataset_test.label, test_predictions)
```

```
2204.9372912887557
```

While ~$2000 absolute error seems large, domain knowledge (and training losses) suggest that this is as good as anyone can get, since a variance of 2K in *listing price* is completely normal, even for the exact same car. Some dealers will issue a "dealer discount" of 1-3K (sometimes even more) on the listing to attract potential buyers and then add those 1-3K back as "extra equipment", "dealer fees", "transportation fees", etc., etc., etc., when you start discussing payment with them (though you can haggle some of this away). Other dealers – usually the bigger ones – are more honest and will not do this. I have once had a scammy dealer try to pull a 3K (?!) transportation fee on me for a car that got shipped from New York to New Jersey (where they were located)!

We also tried adding the column `dealer_name` into the inputs, and the testing mean absolute error immediately went down to ~$1800... However, from the perspective of someone trying to get a car, they will not know what dealer it will be from and so the column `dealer_name` should logically not be a predictor.

End result of Section 2:

A LGBM model has been trained and is stored in the folder `Model` along with the label encoders used to encode the categorical features in the data. This model and encoders are then accessed by Section 4.

Description and Key Features of Section 4: Interactive Jupyter Notebook Demo.

Relevant code in repository:

- Section4_InteractiveDemo.ipynb
- Section4_InteractiveDemo.html

Note: The .html file is a fully rendered version of the Jupyter Notebook and contains the exact same code. This is done to allow graders to easily view the work completed. If desired, the Jupyter Notebook can be run to verify results, etc.

Basic overview:

A database without a way to access it is no database at all! In our interactive Jupyter Notebook demo, we allow any user with the database on their system to run arbitrary SQL queries and use the trained model in Section 3 to make any prediction they want. This notebook accesses the database with the following functions:

```
def init():
    """Connect to MySQL Server, sets globals"""
    # globals
    global mysql_connection
    global mysql_cursor
    global lgbm_model
    global encoder_dict

    # sanity check
    if mysql_connection is not None:
        print("Error (init): Global mysql_connection must be None when connecting, did you call end?")

    # connect to MySQL Server
    # NOTE: For demo purposes, at present, the password for MySQL server on my local machine
    #       is temporarily set to "insecure_password"
    mysql_connection = pymysql.connect(host="localhost", user="root", password = "insecure_password", db="CarMin")
    mysql_cursor = mysql_connection.cursor()

def end():
    """Close MySQL connection, uses globals"""
    # globals
    global mysql_connection
    global mysql_cursor
    global lgbm_model
    global encoder_dict

    # sanity check
    if mysql_connection is None:
        print("Error (end): Global mysql_connection is None, did you call init?")
        return

    # close pymysql varaibles
    mysql_cursor.close()
    mysql_connection.close()

    # maintain globals
    mysql_connection = None
    mysql_cursor = None
```

Furthermore, this notebook is able to access the model trained in Section 3 since it was stored on file in the folder `Model`. Here is an example of using the model to predict the prices on a few cars (note that only partial information is provided to the model):

```
init()

print_predicted_price(make_name="Chevrolet", model_name="Camaro", production_year=2018, odometer=40000)
print_predicted_price(make_name="Dodge", model_name="Challenger", production_year=2020, odometer=35000)
print_predicted_price(make_name="Volkswagen", model_name="Golf", production_year=2023, odometer=25000)

end()
```

```
The predicted price of the input vehicle is: 29768.50316614245 dollars.
The predicted price of the input vehicle is: 34032.006530875995 dollars.
The predicted price of the input vehicle is: 32662.41339728704 dollars.
```

In fact, these prices look very similar to present listings for these cars with these conditions on CarFax! Finally, here is an example of users executing arbitrary SQL queries:

```
init()

# example to joing make/model information with specific car instances
execute_sql_query("""

SELECT make_name, model_name, production_year, odometer, exterior_color
FROM Car LEFT OUTER JOIN MMYT USING (mmyt_id)
WHERE make_name = "Ford" AND model_name = "Focus"
LIMIT 5;

""")

end()
```

```
Ford Focus 2013 100639.0 UNKNOWN
Ford Focus 2013 102699.0 WHITE
Ford Focus 2013 108966.0 BLACK
Ford Focus 2013 150394.0 BLACK
Ford Focus 2013 79369.0 GRAY
```

Description and Key Features of Section 5: Plotting and EDA using R.

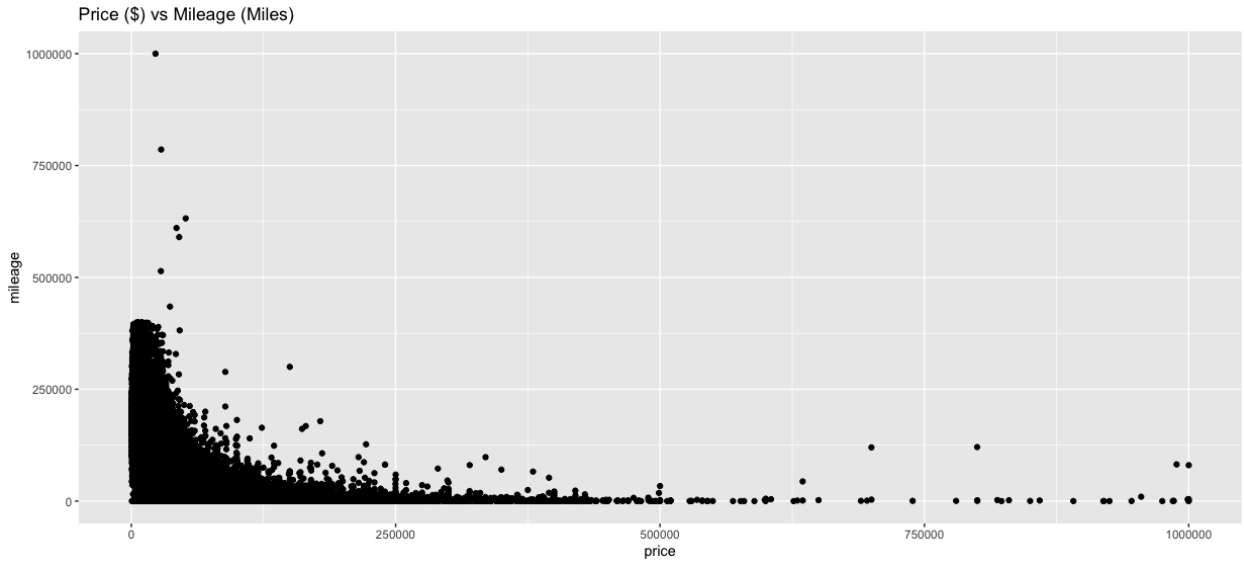
Relevant code in repository:

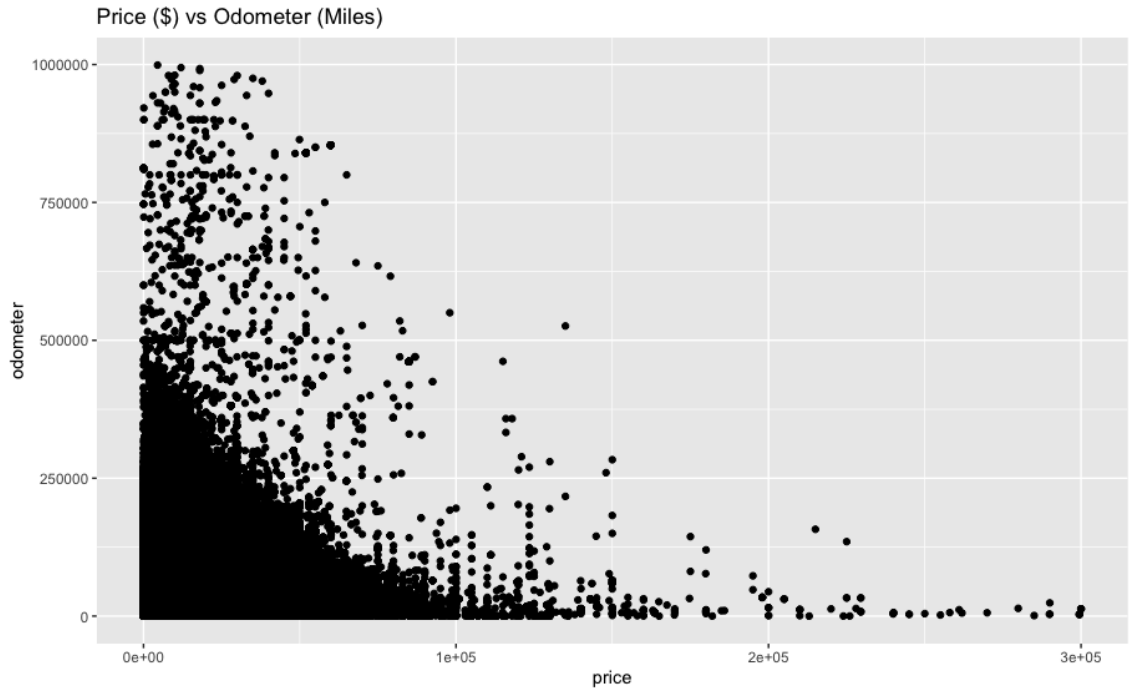
- The folder `Section5_EDA`

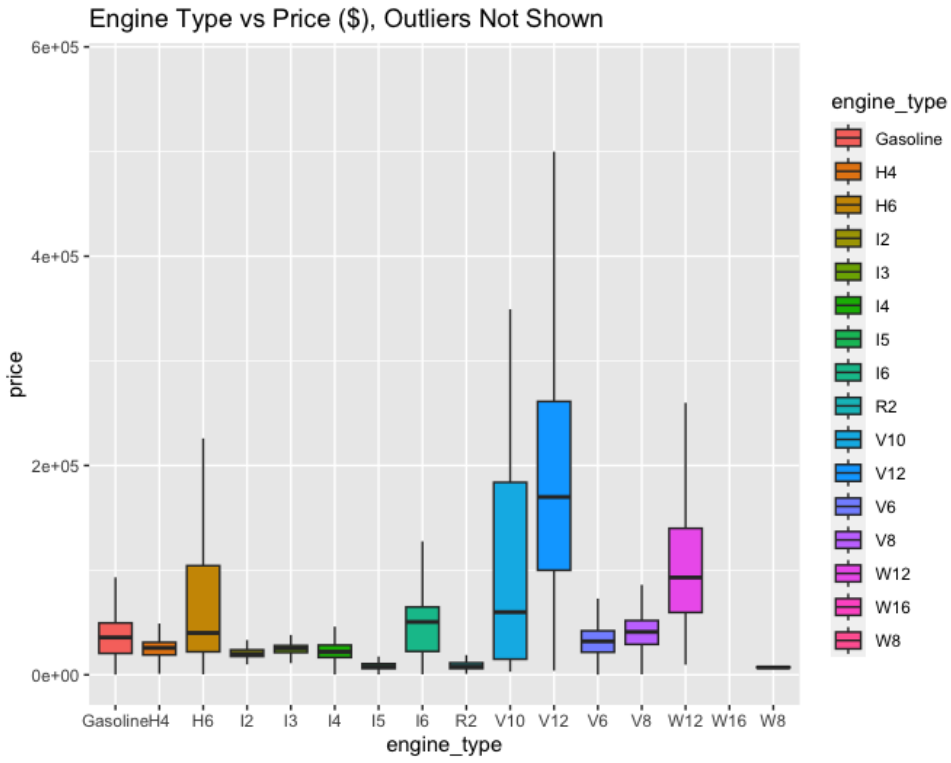
Extra credit attempted: We hope that this entire section may be assessed under the "extraordinary extra features" section of the rubric for extra credit if deemed deserving!

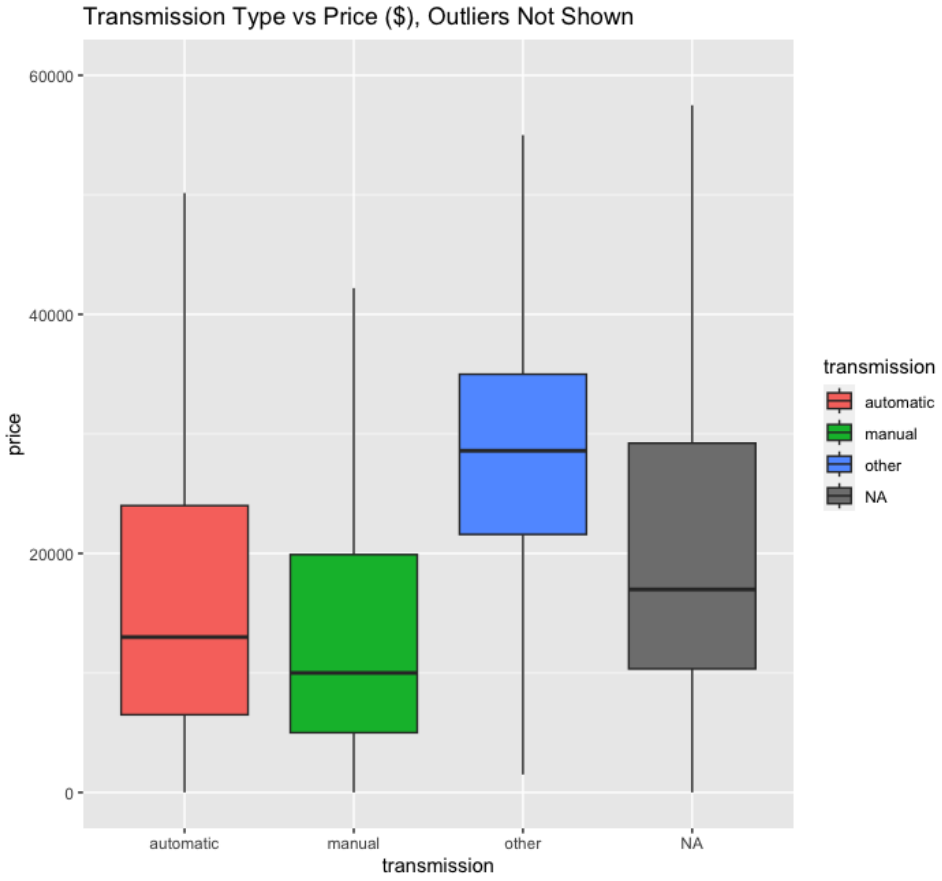
Basic overview:

We visualized the distribution and relationships between price, mileage, and engine/transmission type since all three factors play a recognizable role in purchasing a used car. Boxplots of price and mileage showed that cars from the first dataset had an average price close to 25,000 dollars and an average mileage of approximately 10,000 miles. Additional plots showed that price did not vary significantly (all had an average price between 50,000 and 75,000 dollars) between cars with different engine types except for cars with W12 or V12 engines. W12 cars had an average price of around 100,000 dollars and V12 cars had an average price of around 200,000 dollars. When comparing price versus mileage, there was a clear exponential decay relationship between the two variables. Similar analysis was performed on dataset 2 which contained almost 500,000 entries. The distribution of price and mileage differed from the distributions in dataset 1; the cars in dataset 2 had an average price of under 25,000 and an average total miles of 100,000. When factoring transmission type into distribution of price, cars with unusual transmission (listed as other) had an average higher price (30,000 dollars compared to 18,000 dollars) than cars with standard (automatic or manual) transmissions or unknown transmissions. Furthermore, cars with standard transmissions had an average of 100,000 miles while cars with unusual transmissions had noticeably fewer total miles (around 40,000). Finally, price and total miles displayed the same relationship as price and mileage from dataset 1. A few of our plots are shown below.


Price ($) vs Mileage (Miles)
mileage
price
1000000
750000
500000
250000
0
0
250000
500000
750000
1000000


Price ($) vs Odometer (Miles)
odometer
price
1000000
750000
500000
250000
0
0e+00
1e+05
2e+05
3e+05


Engine Type vs Price ($), Outliers Not Shown
6e+05
4e+05
2e+05
0e+00
price
Gasoline
H4
H6
I2
I3
I4
I5
I6
R2
V10
V12
V6
V8
W12
W16
W8
engine_type
engine_type
Gasoline
H4
H6
I2
I3
I4
I5
I6
R2
V10
V12
V6
V8
W12
W16
W8


Transmission Type vs Price ($), Outliers Not Shown
60000
40000
20000
0
price
automatic
manual
other
NA
transmission
transmission
automatic
manual
other
NA