ETL Report

Dennis Kelly, Robert Stewart, Ryan-Arnold Gamilo

Introduction

How well do past auto industry sales predict future sales? What are the current state of the market? Are there any discernible trends within vehicle sales data? Disrupted by COVID, the vehicle industry remains in a volatile state, and anyone seeking to forecast its future had best be able to answer these questions. Our group wanted to do so, and moreover, do so in an empirically rigorous way. We collected and analyzed numerous datasets to formulate our answers to these questions.

However, before the analysis, we first had to find, clean, and prepare our data to make sure it was sensible and in a suitable format. This document describes where all our datasets were found, as well as the transformations we applied to them before loading them into our database, which served as the source for all future analyses.

Data Sources

U.S. Bureau of Economic Analysis, Motor Vehicle Retail Sales: Domestic Autos [DAUTOSAAR], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/DAUTOSAAR, September 22, 2022.

U.S. Bureau of Economic Analysis, Motor Vehicle Retail Sales: Domestic Light Weight Trucks [DLTRUCKSSAAR], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/DLTRUCKSSAAR, September 22, 2022.

U.S. Bureau of Economic Analysis, Motor Vehicle Retail Sales: Foreign Autos [FAUTOSAAR], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/FAUTOSAAR, September 22, 2022.

U.S. Bureau of Economic Analysis, Motor Vehicle Retail Sales: Foreign Light Weight Trucks [FLTRUCKSSAAR], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/FLTRUCKSSAAR, September 22, 2022.

U.S. Bureau of Economic Analysis, Motor Vehicle Retail Sales: Heavy Weight Trucks [HTRUCKSSAAR], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/HTRUCKSSAAR, September 21, 2022.

United States Census Bureau. American Community Survey: S0802: MEANS OF TRANSPORTATION TO WORK BY SELECTED CHARACTERISTICS, September 21, 2022.

United States Census Bureau. American Community Survey: S0501: SELECTED CHARACTERISTICS OF THE NATIVE AND FOREIGN-BORN POPULATIONS, September 21, 2022.

Cars.com. https://www.cars.com/research/, September 21, 2022

Carsalesbase.com. carsalesbase.com, September 21, 2022

Edmunds's. Most Popular Cars in America https://www.edmunds.com/most-popular-cars/, September 21, 2022

Extraction

FRED Data

1. We used the Python requests library to retrieve several different FRED series via their API.
2. First, in a Jupyter notebook, we import the pandas and requests libraries, and then create a new dataframe, called sales_df, which has one empty column called "date".
3. Next, we create a list of series IDs for the FRED series we wished to request. In order, these were DAUTOSAAR, DLTRUCKSSAAR, FAUTOSAAR, FLTRUCKSSAAR, and HTRUCKSSAAR, corresponding to monthly sales of domestic automobiles, domestic light trucks, foreign automobiles, foreign light trucks, and heavy trucks respectively.
4. Iterating over this loop, we first sent a GET request to

   https://api.stlouisfed.org/fred/series/observations?series_id=DAUTOSAAR&api_key=YOUR_KEY_GOES_HERE&file_type=json

5. Note you must replace "YOUR_KEY_GOES_HERE" in the link above with a valid FRED API key.
6. This returns a requests.Response Python object, and we use that object's json method of this object to convert it into a dictionary.
7. This dictionary has an "observations" key, and the corresponding value is itself a list of dictionaries recording the data we want.
8. We use pandas.DataFrame to convert this list of dictionaries into a dataframe called individual_df, retaining only the "date" and "value" fields of each constituent dictionary in the list.
9. As the last step in each loop iteration, we merge the sales_df, created outside the loop, with individual_df, created inside. We do this by calling the merge method on sales_df, which puts it on the left, and passing it individual_df. The merge is performed on the identity of the "date" field in both sales_df and individual_df.
10. The loop iterates over the remaining series IDs, changing the relevant portion of the url each time.
11. At the conclusion of the loop, sales_df now has 6 columns and 668 rows, though this latter number could change as the data continues to be updated.
12. We got one more series from FRED: historical data on gas prices. Using the requests library, we sent a GET request to

https://api.stlouisfed.org/fred/series/observations?series_id=GASREGCOVW&api_key={api_key}&file_type=json

13. We then called the json method of the requests.Response object to obtain a dictionary, which as before has an "observations" key with a list of dictionaries as the associated value.
14. We used the pandas.DataFrame function to convert this list of dictionaries into a pandas dataframe, again retaining only the "date" and "value" keys in the final dataframe.

Census Data

1. We used the Python requests library to retrieve data from the US Census Bureau's American Community Survey via their API.
2. First, in a new Jupyter notebook, we imported the numpy, pandas, and requests Python libraries
3. We sent a GET request to

   https://api.census.gov/data/2021/acs/acs1/subject?get=NAME,S0802_C01_001E,S0802_C01_070E,S0802_C01_090E,S0802_C02_070E,S0802_C02_090E,S0802_C03_070E,S0802_C03_090E,S0802_C04_070E,S0802_C04_090E&for=state:*&key=YOUR_KEY_GOES_HERE

4. Note you must replace the YOUR_KEY_GOES_HERE portion of the URL above with a valid US Census API key to do this in Python.
5. The URL above is somewhat opaque, but each of the comma separated values following the get phrase in it asks the census to return the value of a specific field. The names of these fields and the corresponding census tag are given in the table below

S0802_C01_001E	Total number of workers 16+
S0802_C01_070E	Total number of workers 16+ who don't work at home
S0802_C01_090E	Average travel time to work for all workers who don't work at home
S0802_C02_070E	Number of workers 16+ who drive alone to work
S0802_C02_090E	Average travel time to work for workers who drive alone to work
S0802_C03_070E	Number of workers 16+ who carpool to work
S0802_C03_090E	Average travel time to work for workers who carpool to work
S0802_C04_070E	Number of workers 16+ who take public transportation to work

S0802_C04_090E	Average travel time to work for workers who take public transportation to work

6. We use the json method of a requests.Response object to obtain a dictionary containing all of the info returned by the Census API, and then use pandas.DataFrame to yield a pandas dataframe from this dictionary.
7. This dataframe is not in a suitable form for analysis. We applied some transformations to reshape it, described under the Census Data heading in the Transformation section below
8. Then, we sent a GET request to

   https://api.census.gov/data/2021/acs/acs1/subject?get=NAME,S0501_C01_128E,S0501_C01_129E&for=state:*&key=YOUR_KEY_GOES_HERE

9. Again, this URL is somewhat opaque. We give a table recording what each tag represents

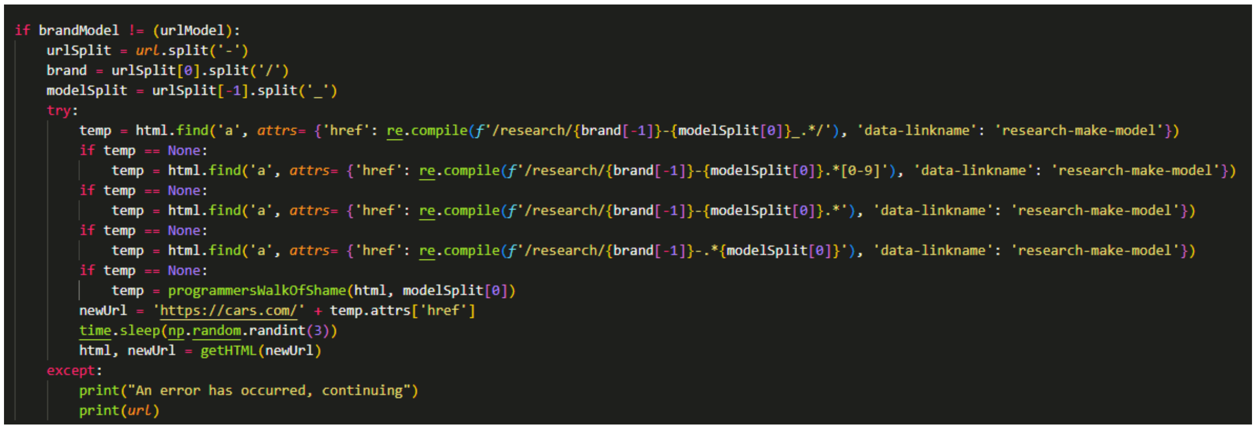
S0501_C01_128E	Percentage of occupied housing units with no access to vehicle
S0501_C01_129E	Percentage of occupied housing units with access to 1 or more vehicles

10. We use the json method of a requests.Response object to yield a dictionary all of the info returned by the Census API, and use pandas to turn this into a dataframe. We retain only the info on "State" and percentage of occupied housing units with access to 1 or more vehicles.
11. Save this dataframe as a csv with pandas. We named this file "state_car_access.csv"

Cars.com Data

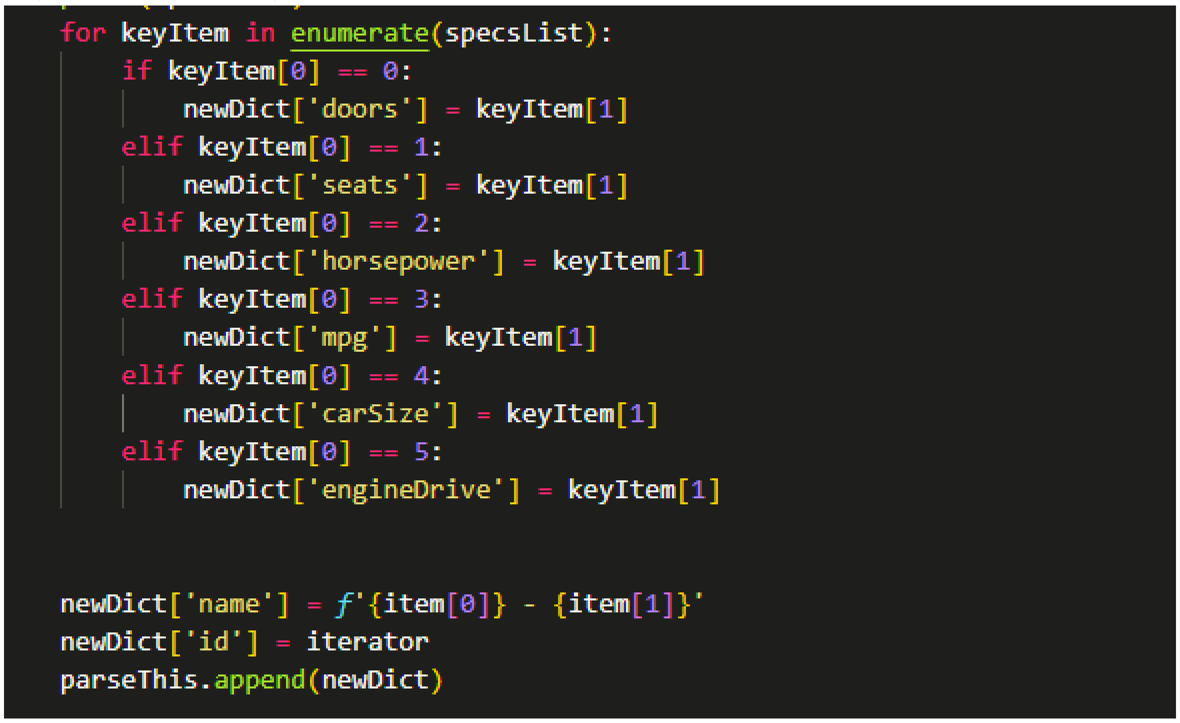
To properly compare the cars gathered, we need to inspect the various aspects of the cars. So, we have to scrape this data in order to compare our cars. Our first step was to look at what the dataset looks like on www.cars.com.

1. First, we pull in the cars data from our original dataset and store it in a pandas data frame.
2. Then, we go to https://www.cars.com/research/?make=&model=&year=
3. Using this we can create a list of the car brands and compare with the car brands from the data frame and create a list of brands that are standardized for the website. All spaces and dashes are converted to underscores. This is also done to the model for each car and stored as a tuple in a list for a loop later.
4. We then looped through each car creating the link for the website. https://www.cars.com/research/{brand_name}-{model_name} This is then passed into a getHTML function. This function does some regular expression testing in order to try and guarantee that the link we build actually finds a car.



```python
if brandModel != (urlModel):
    urlSplit = url.split('-')
    brand = urlSplit[0].split('/')
    modelSplit = urlSplit[-1].split('_')
    try:
        temp = html.find('a', attrs= {'href': re.compile(f'/research/{brand[-1]}-{modelSplit[0]}_.*/'), 'data-linkname': 'research-make-model'})
        if temp == None:
            temp = html.find('a', attrs= {'href': re.compile(f'/research/{brand[-1]}-{modelSplit[0]}.*[0-9]'), 'data-linkname': 'research-make-model'})
        if temp == None:
            temp = html.find('a', attrs= {'href': re.compile(f'/research/{brand[-1]}-{modelSplit[0]}.*'), 'data-linkname': 'research-make-model'})
        if temp == None:
            temp = html.find('a', attrs= {'href': re.compile(f'/research/{brand[-1]}-.*{modelSplit[0]}'), 'data-linkname': 'research-make-model'})
        if temp == None:
            temp = programmersWalkOfShame(html, modelSplit[0])
        newUrl = 'https://cars.com/' + temp.attrs['href']
        time.sleep(np.random.randint(3))
        html, newUrl = getHTML(newUrl)
    except:
        print("An error has occurred, continuing")
        print(url)
```

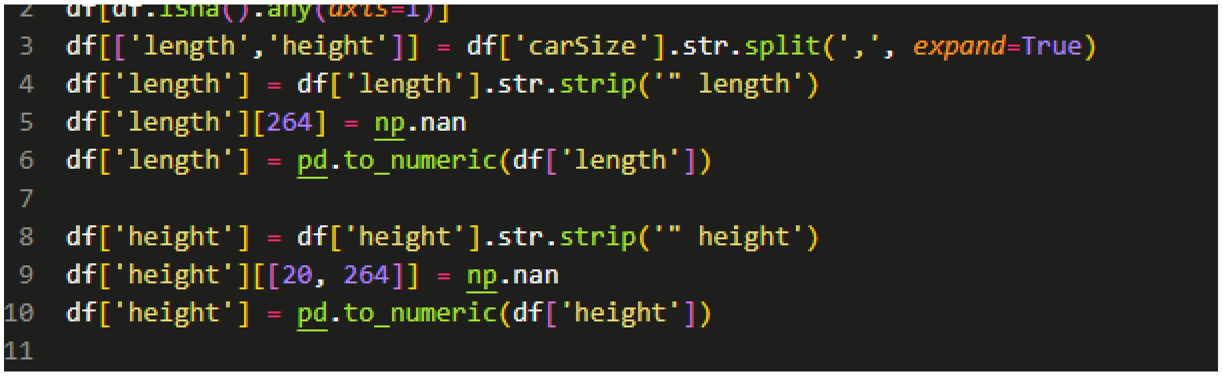
5. With this new url we can look for the first button on the page that corresponds to a year that is not 2023. This way we can guarantee the newest model of the car that still corresponds to our dataset.
6. This allows us to call the final html call to pull the specs of the car off the website and store it in a list of dictionaries. Every page is structured the same, so we can hard define the keys of the dictionary.



```python
    for keyItem in enumerate(specsList):
        if keyItem[0] == 0:
            newDict['doors'] = keyItem[1]
        elif keyItem[0] == 1:
            newDict['seats'] = keyItem[1]
        elif keyItem[0] == 2:
            newDict['horsepower'] = keyItem[1]
        elif keyItem[0] == 3:
            newDict['mpg'] = keyItem[1]
        elif keyItem[0] == 4:
            newDict['carSize'] = keyItem[1]
        elif keyItem[0] == 5:
            newDict['engineDrive'] = keyItem[1]


    newDict['name'] = f'{item[0]} - {item[1]}'
    newDict['id'] = iterator
    parseThis.append(newDict)
```

7. Finally we need to reorganize the dataframe we make out of this list of dictionaries. Step one is to split up the length and height values stored in carSize.

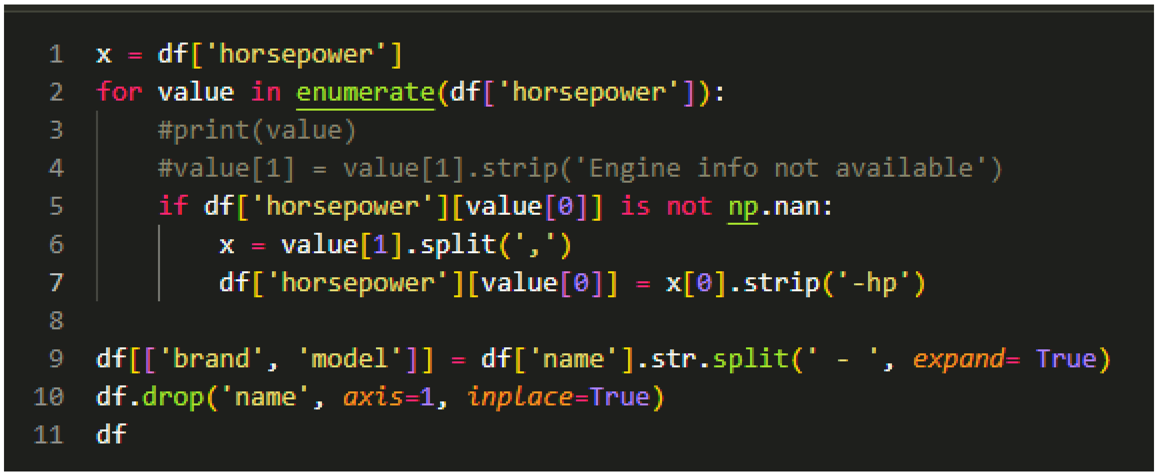


```
df[df.isna().any(axis=1)]
df[['length','height']] = df['carSize'].str.split(',', expand=True)
df['length'] = df['length'].str.strip('" length')
df['length'][264] = np.nan
df['length'] = pd.to_numeric(df['length'])

df['height'] = df['height'].str.strip('" height')
df['height'][[20, 264]] = np.nan
df['height'] = pd.to_numeric(df['height'])

```

8. We then continue to do this with, mpg, doors, and seats. The key here is to handle the points that don't have data. The website uses various forms of "not available" and could be reasonably replaced with nan.
9. Finally we deal with horsepower, because of the nature of the data, horsepower is a combined string that needs to be handled one string at a time.



```
x = df['horsepower']
for value in enumerate(df['horsepower']):
    #print(value)
    #value[1] = value[1].strip('Engine info not available')
    if df['horsepower'][value[0]] is not np.nan:
        x = value[1].split(',')
        df['horsepower'][value[0]] = x[0].strip('-hp')

df[['brand', 'model']] = df['name'].str.split(' - ', expand= True)
df.drop('name', axis=1, inplace=True)
df
```

10. Finally, the data should be cleaned and ready for use. So we can store the data as a csv and move on. We named it "carSpecs.csv"

Carsalesbase.com Data

1. Since we are web scraping, we need to import BeautifulSoup. Also, since we are pattern matching, we need to import re (a regular expression library).
2. On the carsalesbase.com website, we record the links that under the U.S. Sales Dropdown, ignoring the column that says "Top Level". These are the categories of automobiles that we will be looping through.
    a. This is done by getting the <li> tag with an 'id' of 'menu-item-65301' and from there getting all <li> tags with a "class" attribute that follows this regular expression:
        1. 'menu-item menu-item-type-custom menu-item-object-custom menu-item-has-children dropper drop-it menu-item-[0-9]{5}'

    b. These links are stored in a list
3. For each of the categories in this new list, we need to find the correct link with our desired table. In particular, we are looking for an analysis of all cars in the entire year 2021, since there are other links that do analysis by quarter or half.
    1. To do this we look for links that match this regular expression: ‘.*(analysis)*-[0-9]{4}-[A-Za-z]+-?[A-Za-z]*/$’
    2. To make writing data easier, the links with the desired table are stored into a tuple, along with the Category it belongs to.
    3. These tuples are then stored into a list
4. For each of the links in this new list, we scrape the tables by getting each of the <tr> tags inside the <tbody> tags. The sub segment, 2021 sales, and 2020 sales are written into a dictionary.
5. The last two words of the current link tell us the subcategory and are written into the dictionary by splitting on the hyphen and returning the last two items of the list.
6. We write the Category stored in each tuple into the dictionary, which is then stored into a list.
7. The list of dictionaries is then saved as “cars.json”.

Dow Jones U.S. Automobiles Index Data

1. Since we are web scraping, we need to import BeautifulSoup. We will also import confluent_kafka as this is used for catching real-time streams of data.
2. We set up the configuration for the producer, consumer, and topic.
    a. The topic is called DJUSAU, for the name of the index.
    b. The consumer group_id needs to be set to a string. This is done so that the consumer reads the last thing stored on the topic instead of retrieving data from the beginning of the topic every single time it runs.
    c. Otherwise, the configuration of the producer and consumer is the same.
3. Google Finance displays the index on its website:
    a. https://www.google.com/finance/quote/DJUSAU:INDEXDJX?sa=X&ved=2ahUKEwjJx7HXrsT6AhX-E1kFHSJMA2oQ3ecFegQIGRAY
    b. From this link we scrape the value of the index, and write it into a dictionary. We also write the time that this scrape was done into the dictionary.
    c. This dictionary is then sent to the topic.
        i. This is repeated 6 times, with a 10 second pause in between each iteration.
            1. The timing synchronizes with the timing of the data factory, which reruns this every minute.
    d. The consumer then retrieves the information from the topic.
        i. This is then appended to a table in the database
        ii. This is also sent to the blob as a csv file for backup purposes.

Edmund’s Data

1. We pull in data on the best-selling car in each state, from Most Popular Cars in America | Edmunds. We relate this to the Census Data on commutes to investigate any potential associations.
2. As the data is very simple, trying to figure out a web-scrape seemed like overkill. Thus, we made the csv filed by hand, recording in each row the name of a state and the name of the best-selling vehicle in that state. We also record this information for Washington DC and Puerto Rico, yielding 52 total rows.
3. When finished save this CSV file. We named it “state_best_selling_cars.csv”

Transformation

FRED Data

Only a minimal amount of transforming was needed for the FRED; we merged the table with sales data and the table with the gas price data.

1. Specifically, with the dataframe containing the sales data on the left and the dataframe containing the gas data on the right, we used pandas merge functionality to combine these frames. The join method was "outer” and both the left_on and right_on parameters were set to “date”.
2. Save the resulting merged dataframe to a CSV file, which we named “merged_timeseries.csv”

Census Data

We had to reshape the data about commutes returned by the Census. The exact steps are described below.

1. As the result of the API call, we have a commuting information dataframe where each row corresponds to a state, and each row contains the number of workers and mean travel times for all methods of commuting.
2. Instead, we’d like a dataframe where each row corresponds to the unique combination of state and commuting method.
3. First, modifying the overall dataframe, we add columns for the Number of Work from Home workers in each state, which we define as the difference between the Total Workers 16+ and Total Workers 16+ Who Don’t Work at Home fields, and for Other Commuters, defined as the difference of the Total Workers 16+ Who Don’t Work at Home values in each and the sum of the explicitly mentioned commuting methods.
4. We add two more columns corresponding to the average commute time for Work from Home and Other Commuters. Each of these columns consists entirely of NaNs.
5. Now, we break out each commuting method into a separate dataframe by selecting the “State” column, the column giving the number of workers for that commuting method, and the column giving the mean travel time to work for that commuting method. We also add a column to each dataframe giving the name of the commuting method.

6. All told there are now 6 dataframe, one for each commuting method, each with 52 rows. For all 6 dataframes we rename the columns to be “State”, “Number of Workers”, “Mean Transit Time” and “Method”.
7. Finally, we use the pd.concat function to combine all 6 dataframes created in earlier steps into one dataframe, containing all of the same info originally returned by the Census but now in the format we desire.
8. We save this dataframe as a CSV file with pandas. We named it “state_commute_lengths.csv”

Cars.com

1. Cleaning carSpecs.csv
    a. if the word “mazda” appears in a model name, replace it with an empty string
    b. in the engine column, replace any instance of “Engine info not available” with NaN
    c. create a column called sub_segment that combines brand, “ “, and model, as well as replaces any “_” with “ “

Carsalesbase.com

1. Cleaning cars.json
    a. for each sub_segment:
        i. replace “-” with “ “
        ii. replace anything with a slash with every other character in front of it
        iii. if a sub_segment contains mazda, replace it with everything after the space
        iv. remove leading and trailing spaces

<u>Load</u>

Many of the steps above finished by producing files, in either csv or json format, which we subsequently uploaded to a container in Azure Data Lake. We then loaded these into a SQL database using pipelines in an Azure Data Factory, with Python code in Databricks extracting the relevant parts of the files and sending them to the appropriate tables in the SQL database. The Dow Jones U.S. Automobiles Index was also loaded directly into a SQL database via Databricks as part of an automated pipeline in our Data Factory.

<u>Conclusion</u>

This report describes in words the main steps of our ETL process. However, it is often beneficial to see the code executing the operations described. For this reason, we include all the Python code performing ETL work in our GitHub repo.