Experiment 09

Aim:

To perform Exploratory data analysis using Apache Spark and Pandas

Introduction to Big Data and Spark:

Big Data

Big Data is a collection of data that is huge in volume, yet growing exponentially with time. It is data with such large size and complexity that none of traditional data management tools can store it or process it efficiently. Big data is also data but with huge size. The definition of big data is data that contains greater variety, arriving in increasing volumes and with more velocity. This is also known as the three Vs.

The three Vs of big data:

1. Volume
   The amount of data matters. With big data, you'll have to process high volumes of low-density, unstructured data. This can be data of unknown value, such as Twitter data feeds, clickstreams on a web page or a mobile app, or sensor-enabled equipment. For some organizations, this might be tens of terabytes of data. For others, it may be hundreds of petabytes.
2. Velocity
   Velocity is the fast rate at which data is received and (perhaps) acted on. Normally, the highest velocity of data streams directly into memory versus being written to disk. Some internet-enabled smart products operate in real time or near real time and will require real-time evaluation and action.
3. Variety
   Variety refers to the many types of data that are available. Traditional data types were structured and fit neatly in a relational database. With the rise of big data, data comes in new unstructured data types. Unstructured and semistructured data types, such as text, audio, and video, require additional preprocessing to derive meaning and support metadata.

Big Data Mining

Big data mining refers to the collective data mining or extraction techniques that are performed on large sets /volume of data or the big data. Big data mining is primarily done to extract and retrieve desired information or patterns from humongous quantities of data.

This is usually performed on a large quantity of unstructured data that is stored over time by an organization. Typically, big data mining works on data searching, refinement , extraction and comparison algorithms. Big data mining also requires support from underlying computing

devices, specifically their processors and memory, for performing operations / queries on large amounts of data.
Big data mining techniques and processes are also used within big data analytics and business intelligence to deliver summarized targeted and relevant information, patterns and/or relationships between data, systems, processes and more.

Big Data Tools

Big Data requires a set of tools and techniques for analysis to gain insights from it. Big Data is an essential part of almost every organization these days and to get significant results through Big Data Analytics a set of tools is needed at each phase of data processing and analysis.
There are a few factors to be considered while opting for the set of tools i.e., the size of the datasets, pricing of the tool, kind of analysis to be done, and many more.
There are a number of big data tools available in the market such as Hadoop which helps in storing and processing large data, Spark helps in-memory calculation, Storm helps in faster processing of unbounded data, Apache Cassandra provides high availability and scalability of a database, MongoDB provides cross-platform capabilities, so there are different functions of every Big Data tool.

Here is the list of top 10 big data tools –

1. Apache Hadoop
2. Apache Spark
3. Flink
4. Apache Storm
5. Apache Cassandra
6. MongoDB
7. Kafka
8. Tableau
9. RapidMiner
10. R Programming

Spark

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching and optimized query execution for fast queries against data of any size. Simply put, Spark is a fast and general engine for large-scale data processing.

The fast part means that it’s faster than previous approaches to work with Big Data like classical MapReduce. The secret for being faster is that Spark runs on memory (RAM), and that makes the processing much faster than on disk drives.

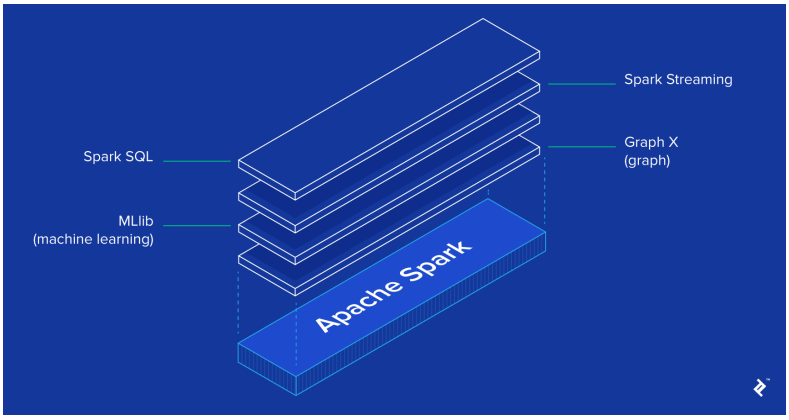
The general part means that it can be used for multiple things like running distributed SQL, creating data pipelines, ingesting data into a database, running Machine Learning algorithms, working with graphs or data streams, and much more.

Features of Spark

1. Fast processing – The most important feature of Apache Spark that has made the big data world choose this technology over others is its speed. Big data is characterized by volume, variety, velocity, and veracity which needs to be processed at a higher speed. Spark contains Resilient Distributed Dataset (RDD) which saves time in reading and writing operations, allowing it to run almost ten to one hundred times faster than Hadoop.
2. Flexibility – Apache Spark supports multiple languages and allows the developers to write applications in Java, Scala, R, or Python.
3. In-memory computing – Spark stores the data in the RAM of servers which allows quick access and in turn accelerates the speed of analytics.
4. Real-time processing – Spark is able to process real-time streaming data. Unlike MapReduce which processes only stored data, Spark is able to process real-time data and is, therefore, able to produce instant outcomes.
5. Better analytics – In contrast to MapReduce that includes Map and Reduce functions, Spark includes much more than that. Apache Spark consists of a rich set of SQL queries, machine learning algorithms, complex analytics, etc. With all these functionalities, analytics can be performed in a better fashion with the help of Spark.

The Spark framework includes:

1. Spark Core as the foundation for the platform
2. Spark SQL for interactive queries
3. Spark Streaming for real-time analytics
4. Spark MLlib for machine learning
5. Spark GraphX for graph processing



Spark Function and Libraries Used:

(Explain the library function used for EDA.)

pyspark.ml module

MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

1. ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
2. Featurization: feature extraction, transformation, dimensionality reduction, and selection
3. Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
4. Persistence: saving and load algorithms, models, and Pipelines
5. Utilities: linear algebra, statistics, data handling, etc.

Correlation

Calculating the correlation between two series of data is a common operation in Statistics. In spark.ml we provide the flexibility to calculate pairwise correlations among many series. The supported correlation methods are currently Pearson's and Spearman's correlation.

Correlation computes the correlation matrix for the input Dataset of Vectors using the specified method. The output will be a DataFrame that contains the correlation matrix of the column of vectors.

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import Correlation

data = [(Vectors.sparse(4, [(0, 1.0), (3, -2.0)]),),
        (Vectors.dense([4.0, 5.0, 0.0, 3.0]),),
        (Vectors.dense([6.0, 7.0, 0.0, 8.0]),),
        (Vectors.sparse(4, [(0, 9.0), (3, 1.0)]),)]
df = spark.createDataFrame(data, ["features"])

r1 = Correlation.corr(df, "features").head()
print("Pearson correlation matrix:\n" + str(r1[0]))

r2 = Correlation.corr(df, "features", "spearman").head()
print("Spearman correlation matrix:\n" + str(r2[0]))
```

Summarizer

We provide vector column summary statistics for Dataframe through Summarizer. Available metrics are the column-wise max, min, mean, sum, variance, std, and number of nonzeros, as well as the total count.

```
from pyspark.ml.stat import Summarizer
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

df = sc.parallelize([Row(weight=1.0, features=Vectors.dense(1.0, 1.0, 1.0)),
                     Row(weight=0.0, features=Vectors.dense(1.0, 2.0, 3.0))]).toDF()

# create summarizer for multiple metrics "mean" and "count"
summarizer = Summarizer.metrics("mean", "count")
```

```
# compute statistics for multiple metrics with weight
df.select(summarizer.summary(df.features, df.weight)).show(truncate=False)

# compute statistics for multiple metrics without weight
df.select(summarizer.summary(df.features)).show(truncate=False)

# compute statistics for single metric "mean" with weight
df.select(Summarizer.mean(df.features, df.weight)).show(truncate=False)
```

pyspark.sql module

PySpark SQL is a module in Spark which integrates relational processing with Spark's functional programming API. We can extract the data by using an SQL query language. We can use the queries same as the SQL language.

If you have a basic understanding of RDBMS, PySpark SQL will be easy to use, where you can extend the limitation of traditional relational data processing. Spark also supports the Hive Query Language, but there are limitations of the Hive database.

Important classes of Spark SQL and DataFrames:

- pyspark.sql.SparkSession Main entry point for DataFrame and SQL functionality.
- pyspark.sql.DataFrame A distributed collection of data grouped into named columns.
- pyspark.sql.Column A column expression in a DataFrame.
- pyspark.sql.Row A row of data in a DataFrame.
- pyspark.sql.GroupedData Aggregation methods, returned by DataFrame.groupBy().
- pyspark.sql.DataFrameNaFunctions Methods for handling missing data (null values).
- pyspark.sql.DataFrameStatFunctions Methods for statistics functionality.
- pyspark.sql.functions List of built-in functions available for DataFrame.
- pyspark.sql.types List of data types available.
- pyspark.sql.Window For working with window functions.

class pyspark.sql.DataFrame

It is a distributed collection of data grouped into named columns. A DataFrame is similar to the relational table in Spark SQL, can be created using various functions in SQLContext.

```
sqlContext.read.csv("...")
```

After creation of dataframe, we can manipulate it using the several domain-specific-languages (DSL) which are predefined functions of DataFrame. Let’s get started with the functions:

select(): The select function helps us to display a subset of selected columns from the entire dataframe we just need to pass the desired column names. Let’s print any three columns of the dataframe using select().

withColumn(): The withColumn function is used to manipulate a column or to create a new column with the existing column. It is a transformation function, we can also change the datatype of any existing column.

groupBy(): The groupBy function is used to collect the data into groups on DataFrame and allows us to perform aggregate functions on the grouped data. This is a very common data analysis operation similar to the groupBy clause in SQL.

orderBy(): The orderBy function is used to sort the entire dataframe based on the particular column of the dataframe. It sorts the rows of the dataframe according to column values. By default, it sorts in ascending order.

split(): The split() is used to split a string column of the dataframe into multiple columns. This function is applied to the dataframe with the help of withColumn() and select().

lit(): The lit function is used to add a new column to the dataframe that contains literals or some constant value.

when(): The when the function is used to display the output based on the particular condition. It evaluates the condition provided and then returns the values accordingly. It is a SQL function that supports PySpark to check multiple conditions in a sequence and return the value. This function similarly works as if-then-else and switch statements.

filter(): The filter function is used to filter data in rows based on the particular column values.

isNull()/isNotNull(): These two functions are used to find out if there is any null value present in the DataFrame. It is the most essential function for data processing. It is the major tool used for data cleaning.

Code and Observation:

Dataset: Car Features and MSRP
Preprocessing

Converting the data type of desired columns into IntegerType

```
from pyspark.sql import functions as f
from pyspark.sql.types import IntegerType

numeric_columns = ['Year','Engine HP','Engine Cylinders','Number of
Doors','highway MPG','city mpg','Popularity','MSRP']
for column in numeric_columns:
```

```
    df = df.withColumn(column,f.col(column).cast(IntegerType()))
df.printSchema()
```

```
root
 |-- Make: string (nullable = true)
 |-- Model: string (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Engine Fuel Type: string (nullable = true)
 |-- Engine HP: integer (nullable = true)
 |-- Engine Cylinders: integer (nullable = true)
 |-- Transmission Type: string (nullable = true)
 |-- Driven_Wheels: string (nullable = true)
 |-- Number of Doors: integer (nullable = true)
 |-- Market Category: string (nullable = true)
 |-- Vehicle Size: string (nullable = true)
 |-- Vehicle Style: string (nullable = true)
 |-- highway MPG: integer (nullable = true)
 |-- city mpg: integer (nullable = true)
 |-- Popularity: integer (nullable = true)
 |-- MSRP: integer (nullable = true)
```

Handling Missing Values

```
#Count the missing values in every column
for col in df.columns:
  print(df.filter(df[col].isNull()).count(), '\t', col)
```

```
0       Make
0       Model
0       Year
3       Engine Fuel Type
69      Engine HP
30      Engine Cylinders
0       Transmission Type
0       Driven_Wheels
6       Number of Doors
0       Vehicle Size
0       Vehicle Style
0       highway MPG
0       city mpg
0       Popularity
0       MSRP
```

```
#Filling missing values with means
from pyspark.sql.functions import avg

def mean_of_pyspark_columns(df, numeric_cols, verbose=False):
    col_with_mean=[]
    for col in numeric_cols:
        mean_value = df.select(avg(df[col]))
        avg_col = mean_value.columns[0]
        res = mean_value.rdd.map(lambda row : row[avg_col]).collect()

        if (verbose==True): print(mean_value.columns[0], "\t", res[0])
        col_with_mean.append([col, res[0]])
    return col_with_mean
```

```
#Fill missing values for mean
from pyspark.sql.functions import when, lit

def fill_missing_with_mean(df, numeric_cols):
    col_with_mean = mean_of_pyspark_columns(df, numeric_cols)

    for col, mean in col_with_mean:
        df = df.withColumn(col, when(df[col].isNull()==True,
        lit(mean)).otherwise(df[col]))

    return df
numeric_cols=['Engine HP','Engine Cylinders']
df = fill_missing_with_mean(df, numeric_cols)
#Filling missing values with mode
def mode_of_pyspark_columns(df, cat_col_list, verbose=False):
    col_with_mode=[]
    for col in cat_col_list:
        #Filter null
        df = df.filter(df[col].isNull()==False)
        #Find unique_values_with_count
        unique_classes = df.select(col).distinct().rdd.map(lambda x:
x[0]).collect()
        unique_values_with_count=[]
        for uc in unique_classes:
             unique_values_with_count.append([uc,
df.filter(df[col]==uc).count()])
        #sort unique values w.r.t their count values
        sorted_unique_values_with_count= sorted(unique_values_with_count, key
= lambda x: x[1], reverse =True)

        if (verbose==True): print(col, sorted_unique_values_with_count, " and
mode is ", sorted_unique_values_with_count[0][0])
        col_with_mode.append([col, sorted_unique_values_with_count[0][0]])
    return col_with_mode

#Fill missing values for mode
from pyspark.sql.functions import when, lit

def fill_missing_with_mode(df, cat_col_list):
    col_with_mode =mode_of_pyspark_columns(df, cat_col_list)

    for col, mode in col_with_mode:
```

```
        df = df.withColumn(col, when(df[col].isNull()==True,
        lit(mode)).otherwise(df[col]))

    return df
cat_col_list=['Engine Fuel Type']
df = fill_missing_with_mode(df, cat_col_list)

#Count the missing values in every column
for col in df.columns:
  print(df.filter(df[col].isNull()).count(), '\t', col)
```

```
0       Make
0       Model
0       Year
0       Engine Fuel Type
0       Engine HP
0       Engine Cylinders
0       Transmission Type
0       Driven_Wheels
6       Number of Doors
0       Vehicle Size
0       Vehicle Style
0       highway MPG
0       city mpg
0       Popularity
0       MSRP
```

Handling Outliers

```
def find_outliers(df):

    # Identifying the numerical columns in a spark dataframe
    numeric_columns = [column[0] for column in df.dtypes if column[1]=='int']

    # Using the `for` loop to create new columns by identifying the outliers
for each feature
    for column in numeric_columns:

        less_Q1 = 'less_Q1_{}'.format(column)
        more_Q3 = 'more_Q3_{}'.format(column)
        Q1 = 'Q1_{}'.format(column)
        Q3 = 'Q3_{}'.format(column)

        # Q1 : First Quartile ., Q3 : Third Quartile
        Q1 = df.approxQuantile(column,[0.25],relativeError=0)
        Q3 = df.approxQuantile(column,[0.75],relativeError=0)

        # IQR : Inter Quantile Range
```

```
        IQR = Q3[0] - Q1[0]

        #selecting the data, with -1.5*IQR to + 1.5*IQR., where param = 1.5
default value
        less_Q1 =  Q1[0] - 1.5*IQR
        more_Q3 =  Q3[0] + 1.5*IQR

        isOutlierCol = 'is_outlier_{}'.format(column)

        df = df.withColumn(isOutlierCol,f.when((df[column] > more_Q3) |
(df[column] < less_Q1), 1).otherwise(0))


    # Selecting the specific columns which we have added above, to check if
there are any outliers
    selected_columns = [column for column in df.columns if
column.startswith("is_outlier")]

    # Adding all the outlier columns into a new colum "total_outliers", to
see the total number of outliers
    df = df.withColumn('total_outliers',sum(df[column] for column in
selected_columns))

    # Dropping the extra columns created above, just to create nice
dataframe., without extra columns
    df = df.drop(*[column for column in df.columns if
column.startswith("is_outlier")])

    return df
new_df_with_no_outliers = new_df.filter(new_df['total_Outliers']<=1)
new_df_with_no_outliers = new_df_with_no_outliers.select(*df.columns)
new_df = find_outliers(df)
df = new_df_with_no_outliers
new_df.show()
```

Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Vehicle Size	Vehicle Style	highway MPG	city mpg	Popularity	MSRP
BMW	1 Series M	2011	premium unleaded ...	335.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	26	19	3916	46135
BMW	1 Series	2011	premium unleaded ...	300.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	28	19	3916	40650
BMW	1 Series	2011	premium unleaded ...	300.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	20	3916	36350
BMW	1 Series	2011	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	18	3916	29450
BMW	1 Series	2011	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	28	18	3916	34500
BMW	1 Series	2012	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	18	3916	31200
BMW	1 Series	2012	premium unleaded ...	300.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	26	17	3916	44100
BMW	1 Series	2012	premium unleaded ...	300.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	20	3916	39300
BMW	1 Series	2012	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	28	18	3916	36900
BMW	1 Series	2013	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	27	18	3916	37200
BMW	1 Series	2013	premium unleaded ...	300.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	20	3916	39600
BMW	1 Series	2013	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	19	3916	31500
BMW	1 Series	2013	premium unleaded ...	300.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	28	19	3916	44400
BMW	1 Series	2013	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	28	19	3916	37200
BMW	1 Series	2013	premium unleaded ...	230.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	19	3916	31500
BMW	1 Series	2013	premium unleaded ...	320.0	6.0	MANUAL	rear wheel drive	2	Compact	Convertible	25	18	3916	48250
BMW	1 Series	2013	premium unleaded ...	320.0	6.0	MANUAL	rear wheel drive	2	Compact	Coupe	28	20	3916	43550
Audi	100	1992	regular unleaded	172.0	6.0	MANUAL	front wheel drive	4	Midsize	Sedan	24	17	3105	2000
Audi	100	1992	regular unleaded	172.0	6.0	MANUAL	front wheel drive	4	Midsize	Sedan	24	17	3105	2000
Audi	100	1992	regular unleaded	172.0	6.0	AUTOMATIC	all wheel drive	4	Midsize	Wagon	20	16	3105	2000

only showing top 20 rows

EDA

Statistical Measures of Data (Central tendency)

```
df.describe().toPandas().T
```

	0	1	2	3	4
summary	count	mean	stddev	min	max
Make	11690	None	None	Acura	Volvo
Model	11690	745.5822222222222	1490.8280590623795	1 Series	xD
Year	11690	2010.3733105218134	7.548666486104393	1990	2017
Engine Fuel Type	11690	None	None	diesel	regular unleaded
Engine HP	11690	251.04505147577356	108.83210431985397	55.0	1001.0
Engine Cylinders	11690	5.671648979772815	1.740388098307277	3.0	16.0
Transmission Type	11690	None	None	AUTOMATED_MANUAL	UNKNOWN
Driven_Wheels	11690	None	None	all wheel drive	rear wheel drive
Number of Doors	11689	3.429634699289931	0.8841314633993796	2	4
Vehicle Size	11690	None	None	Compact	Midsize
Vehicle Style	11690	None	None	2dr Hatchback	Wagon
highway MPG	11690	26.04328485885372	6.746006685374115	12	354
city mpg	11690	18.952266894781864	4.908865444900392	7	44
Popularity	11690	1530.8038494439693	1422.0136911220768	2	5657
MSRP	11690	40806.753378956375	60573.09270843111	2000	2065902

```
from pyspark.sql.functions import mean, col
df_stats = df.select(
    mean(col('Engine HP')).alias('Mean Engine HP'),
    mean(col('Engine Cylinders')).alias('Mean Engine Cylinders'),
```

```
    mean(col('Number of Doors')).alias('Mean Number of Doors'),
    mean(col('highway MPG')).alias('Mean highway MPG'),
    mean(col('city mpg')).alias('Mean city mpg'),
    mean(col('Popularity')).alias('Mean Popularity'),
    mean(col('MSRP')).alias('Mean MSRP'),
).collect()
for i in df_stats:
  row = i.asDict()
  for k in row:
    print(k," - ", row[k])
```

```
Mean Engine HP  -  251.04505147577356
Mean Engine Cylinders  -  5.671648979772815
Mean Number of Doors  -  3.429634699289931
Mean highway MPG  -  26.04328485885372
Mean city mpg  -  18.952266894781864
Mean Popularity  -  1530.8038494439693
Mean MSRP  -  40806.753378956375
```

```
print("Median Engine HP - ",df.approxQuantile("Engine HP", [0.5], 0.25))
print("Median Engine Cylinders",df.approxQuantile("Engine Cylinders", [0.5],
0.25))
print("Median Number of Doors",df.approxQuantile("Number of Doors", [0.5],
0.25))
print("Median highway MPG",df.approxQuantile("highway MPG", [0.5], 0.25))
print("Median city mpg",df.approxQuantile("city mpg", [0.5], 0.25))
print("Median Popularity",df.approxQuantile("Popularity", [0.5], 0.25))
print("Median MSRP",df.approxQuantile("MSRP", [0.5], 0.25))
```

```
Median Engine HP -  [170.0]
Median Engine Cylinders [4.0]
Median Number of Doors [2.0]
Median highway MPG [22.0]
Median city mpg [16.0]
Median Popularity [549.0]
Median MSRP [21040.0]
```

Inferences:

1. Year, MSRP and MSRP have high difference between their mean and median. Hence, their data is a skewed distribution.
2. Highway MGP and City MPG have very similar mean and median. Hence, their data is a symmetrical distribution.
3. Most number of cars were made in 2016.
4. Most number of cars have 4 doors.

Statistical Measures of Data (Dispersion)

```
quantile = df.approxQuantile(['Engine HP','Engine Cylinders','Number of
Doors','highway MPG','city mpg','Popularity','MSRP'], [0.25, 0.5, 0.75], 0)
quantiles = pd.DataFrame(quantile)
quantiles.columns = ['1st Quartile','Median','3rd Quartile']
quantiles.index = ['Engine HP','Engine Cylinders','Number of Doors','highway
MPG','city mpg','Popularity','MSRP']
quantiles
```

	1st Quartile	Median	3rd Quartile
Engine HP	170.0	230.0	302.0
Engine Cylinders	4.0	6.0	6.0
Number of Doors	2.0	4.0	4.0
highway MPG	22.0	25.0	30.0
city mpg	15.0	18.0	22.0
Popularity	549.0	1385.0	2009.0
MSRP	21030.0	30025.0	42425.0

```
df.agg({
    'Engine HP': 'variance',
    'Engine Cylinders':'variance',
    'Number of Doors':'variance',
    'highway MPG':'variance',
    'city mpg':'variance',
    'Popularity':'variance',
    'MSRP':'variance',
}).show()
```

```
+------------------------+--------------------+------------------+--------------------+-------------------------+--------------------+------------------+
|variance(Number of Doors)|      variance(MSRP)|variance(Engine HP)|variance(Popularity)|variance(Engine Cylinders)|variance(highway MPG)|variance(city mpg)|
+------------------------+--------------------+------------------+--------------------+-------------------------+--------------------+------------------+
|       0.7816884445727287|3.669009560264190...| 11844.426930687576|  2022122.9377386332|        3.0289507327296206|    45.50860619911226| 24.09695995613712|
+------------------------+--------------------+------------------+--------------------+-------------------------+--------------------+------------------+
```

```
df.agg({
    'Engine HP': 'stddev',
    'Engine Cylinders':'stddev',
    'Number of Doors':'stddev',
    'highway MPG':'stddev',
    'city mpg':'stddev',
    'Popularity':'stddev',
    'MSRP':'stddev',
}).show()
```

```
+---------------------+-----------------+-----------------+-----------------+----------------------+------------------+-----------------+
|stddev(Number of Doors)|     stddev(MSRP)| stddev(Engine HP)|stddev(Popularity)|stddev(Engine Cylinders)|stddev(highway MPG)| stddev(city mpg)|
+---------------------+-----------------+-----------------+-----------------+----------------------+------------------+-----------------+
|     0.884131463993796|60573.09270843111|108.83210431985397|1422.0136911220768|      1.740388098307277|  6.746006685374115|4.908865444900392|
+---------------------+-----------------+-----------------+-----------------+----------------------+------------------+-----------------+
```

```
from pyspark.sql.functions import col, skewness, kurtosis
df.select(skewness('MSRP'),kurtosis('MSRP')).show()
```

```
+------------------+-----------------+
|    skewness(MSRP)|   kurtosis(MSRP)|
+------------------+-----------------+
|11.711088325398018|265.4637541011474|
+------------------+-----------------+
```

Inferences:

1. Almost all of the cars manufactured have four doors, as both median and third quartile are equal to 4.
2. Manufactured cars give better Highway MPG as compared to city MPG.
3. The standard deviation values of the dataset show that Engine HP, Popularity and MSRP have the most spread out values among all cars manufactured.
4. MSRP is positively skewed

<u>Correlation Analysis</u>

```
import pandas as pd
from pyspark.mllib.stat import Statistics

features = df.select(numeric_columns).rdd.map(lambda row: row[0:])

corr_mat=Statistics.corr(features, method="pearson")

corr_df = pd.DataFrame(corr_mat,index=numeric_columns,
columns=numeric_columns)
corr_df
```

	Year	Engine HP	Engine Cylinders	Number of Doors	highway MPG	city mpg	Popularity	MSRP
Year	1.000000	0.357434	-0.025484	NaN	0.289862	0.268064	0.094097	0.227169
Engine HP	0.357434	1.000000	0.782573	NaN	-0.422038	-0.551605	0.052245	0.663127
Engine Cylinders	-0.025484	0.782573	1.000000	NaN	-0.625772	-0.734425	0.051207	0.548954
Number of Doors	NaN	NaN	NaN	1.0	NaN	NaN	NaN	NaN
highway MPG	0.289862	-0.422038	-0.625772	NaN	1.000000	0.833835	-0.051463	-0.212294
city mpg	0.268064	-0.551605	-0.734425	NaN	0.833835	1.000000	-0.058781	-0.285327
Popularity	0.094097	0.052245	0.051207	NaN	-0.051463	-0.058781	1.000000	-0.043476
MSRP	0.227169	0.663127	0.548954	NaN	-0.212294	-0.285327	-0.043476	1.000000

```
# removing NaN values
corr_df = corr_df.drop('Number of Doors', axis=1)
corr_df = corr_df.dropna()
corr_df
```

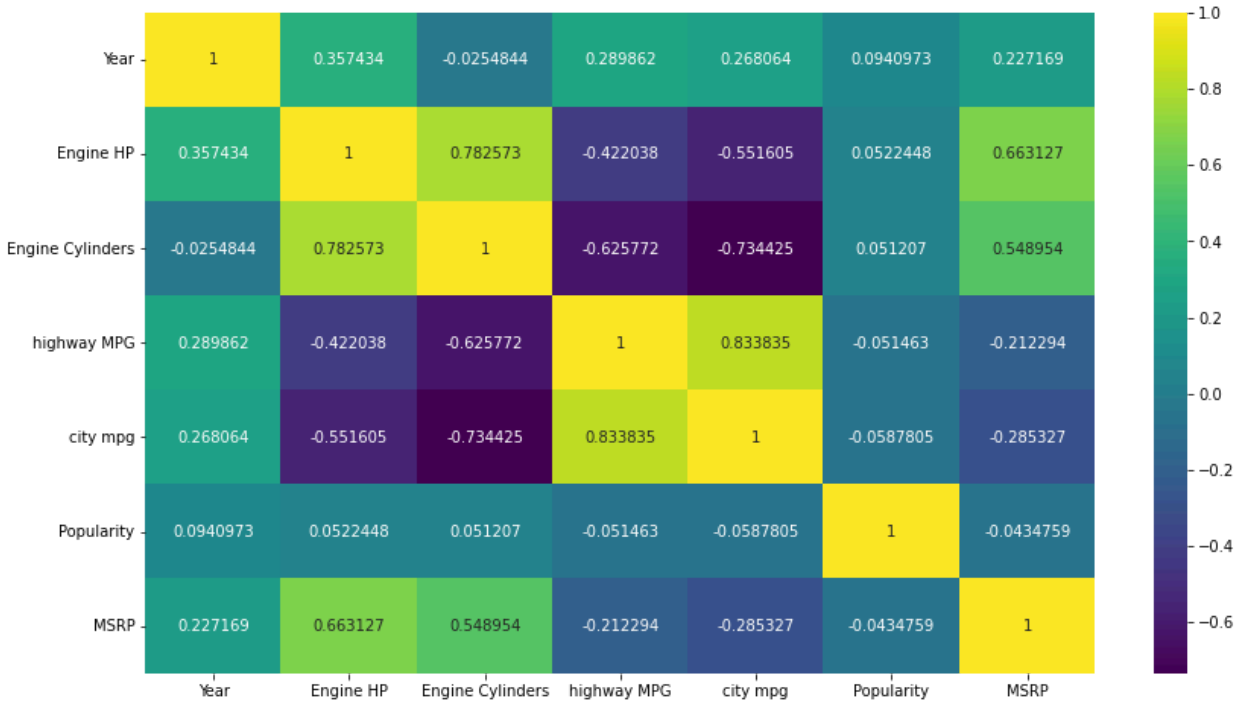
	Year	Engine HP	Engine Cylinders	highway MPG	city mpg	Popularity	MSRP
Year	1.000000	0.357434	-0.025484	0.289862	0.268064	0.094097	0.227169
Engine HP	0.357434	1.000000	0.782573	-0.422038	-0.551605	0.052245	0.663127
Engine Cylinders	-0.025484	0.782573	1.000000	-0.625772	-0.734425	0.051207	0.548954
highway MPG	0.289862	-0.422038	-0.625772	1.000000	0.833835	-0.051463	-0.212294
city mpg	0.268064	-0.551605	-0.734425	0.833835	1.000000	-0.058781	-0.285327
Popularity	0.094097	0.052245	0.051207	-0.051463	-0.058781	1.000000	-0.043476
MSRP	0.227169	0.663127	0.548954	-0.212294	-0.285327	-0.043476	1.000000

```
# get a boolean dataframe where true means that a pair of variables is highly
correlated
highly_correlated_df = (abs(corr_df) > .5) & (corr_df < 1.0)

# get the names of the variables so we can use them to slice the dataframe
correlated_vars_index = (highly_correlated_df==True).any()
correlated_var_names =
correlated_vars_index[correlated_vars_index==True].index

# slice it
highly_correlated_df.loc[correlated_var_names,correlated_var_names]
```

	Engine HP	Engine Cylinders	highway MPG	city mpg	MSRP
Engine HP	False	True	False	True	True
Engine Cylinders	True	False	True	True	True
highway MPG	False	True	False	True	False
city mpg	True	True	True	False	False
MSRP	True	True	False	False	False

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(14,8))
sns.heatmap(corr_df, annot=True, fmt="g", cmap='viridis')
```

Name: Neeraj Rijhwani Roll No: 64 Class:D15C



Observations:

Positive correlation:

1. Cylinders and HP - higher the number of cyclinders higher will be the horse power.
2. Highway mpg and City mpg - higher the highway mpg higher will be the city mpg.

Negative correlation:

1. MPG and Cylinders - higher the number of cyclinders lesser will be the MPG.
2. MPG and HP - higher the number of Power lesser will be the MPG.

No correlation:

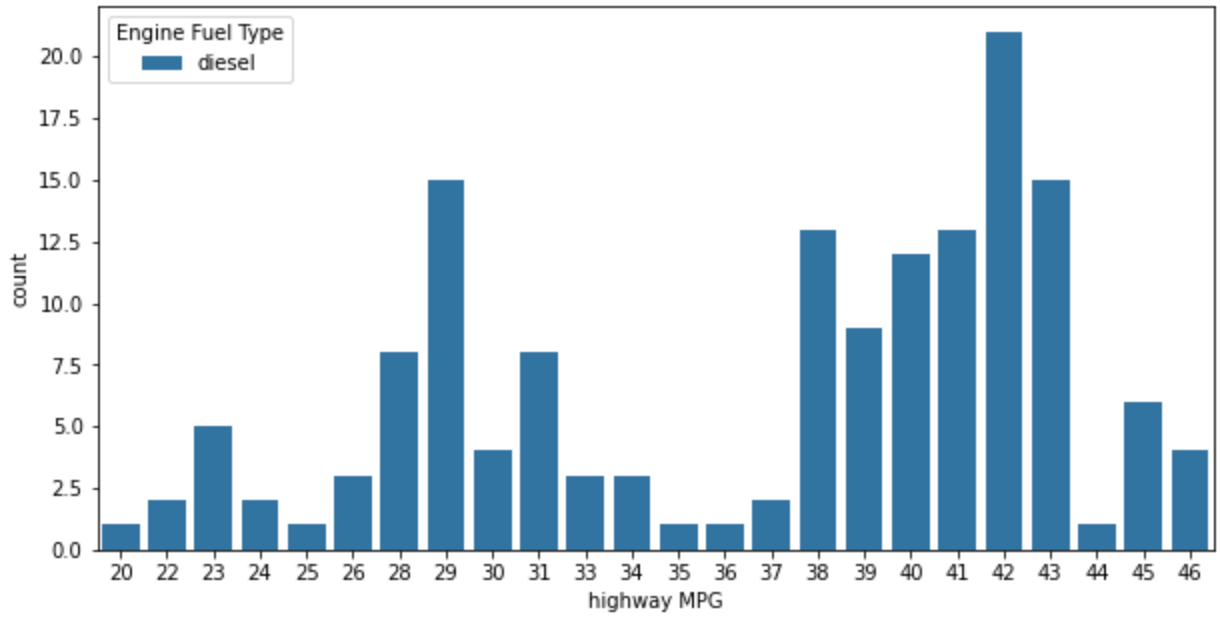
1. Highway mpg and Popularity
2. City mpg and Popularity
3. Cylinders and Popularity

<u>Asking and Answering Questions</u>

Question 1:

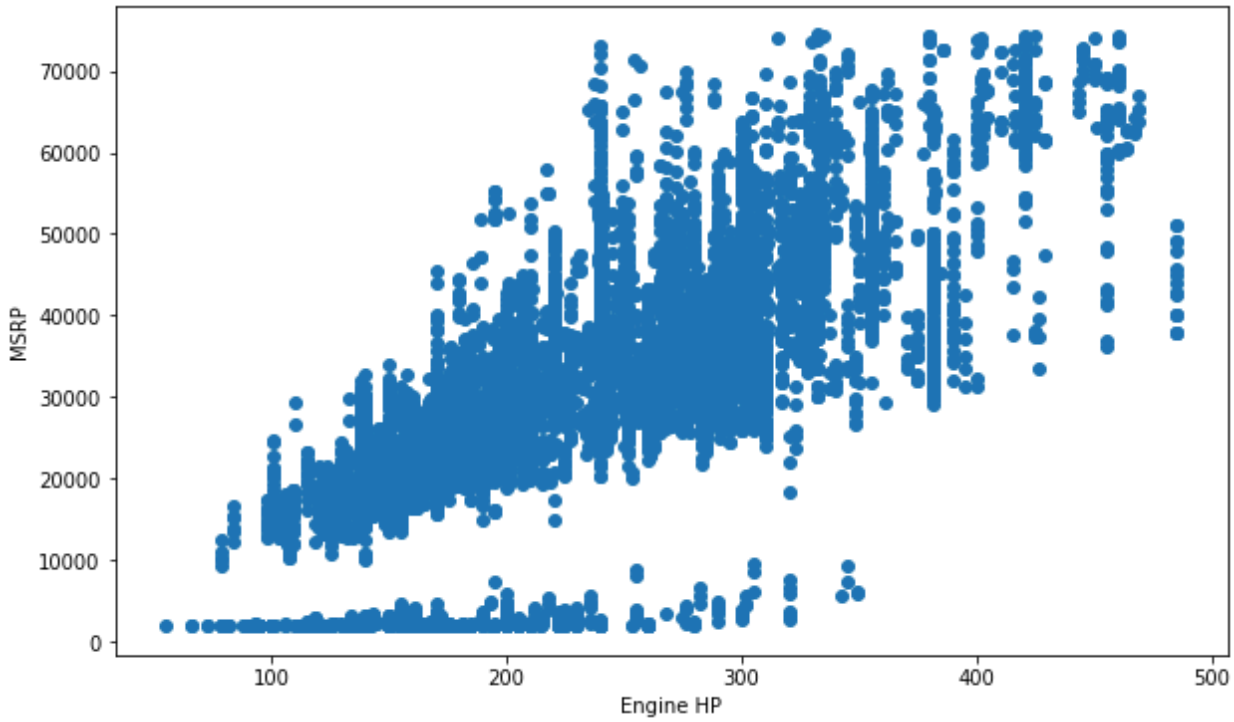
Has use of diesel cars increased or decreased over the years?

Name: Neeraj Rijhwani Roll No: 64 Class:D15C



Answer: Use of diesel cars increased steadily from 2010 to 2015. Thereafter, the usage has gone down drastically.

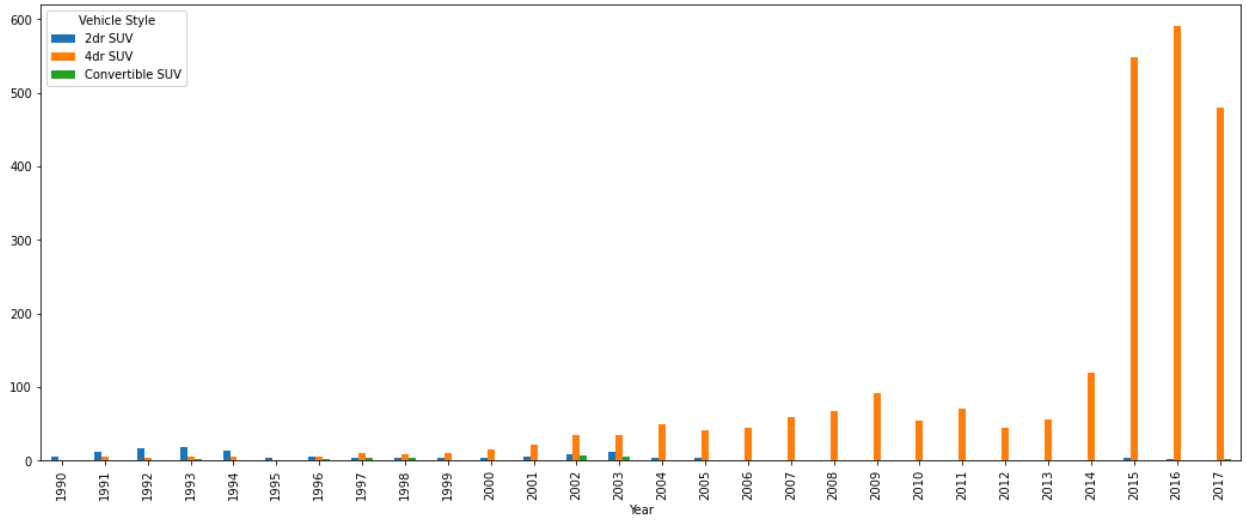
Question 2:
Do higher priced cars come with more powerful engines, that is, higher horsepower?



Answer: Scatterplot above suggests that, yes higher priced cars do provide higher engine horse power.

Name: Neeraj Rijhwani Roll No: 64 Class:D15C

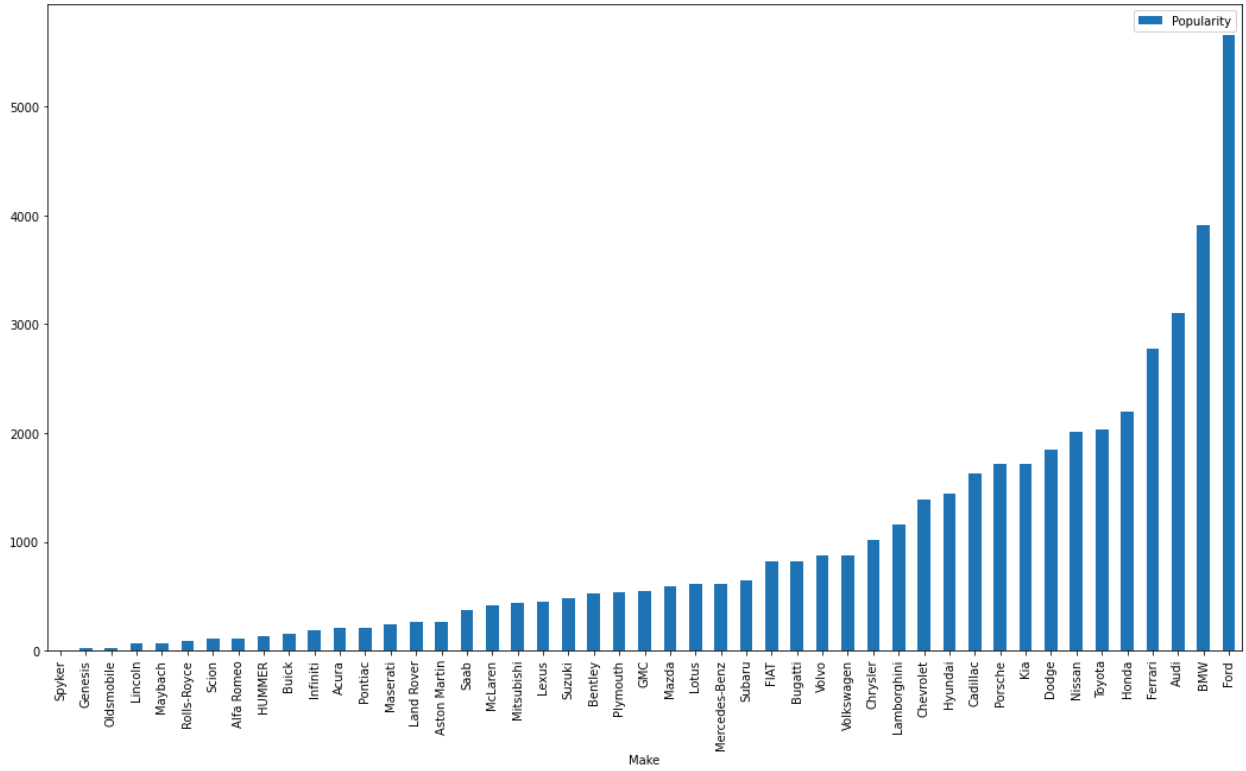
Question 3:
Has production of SUVs increased over the years?



Answer: From the graph we can cleary see that SUV production has gone up significantly as compared to previous decades.

Question 4:
Which brand is least popular and which one is most popular?

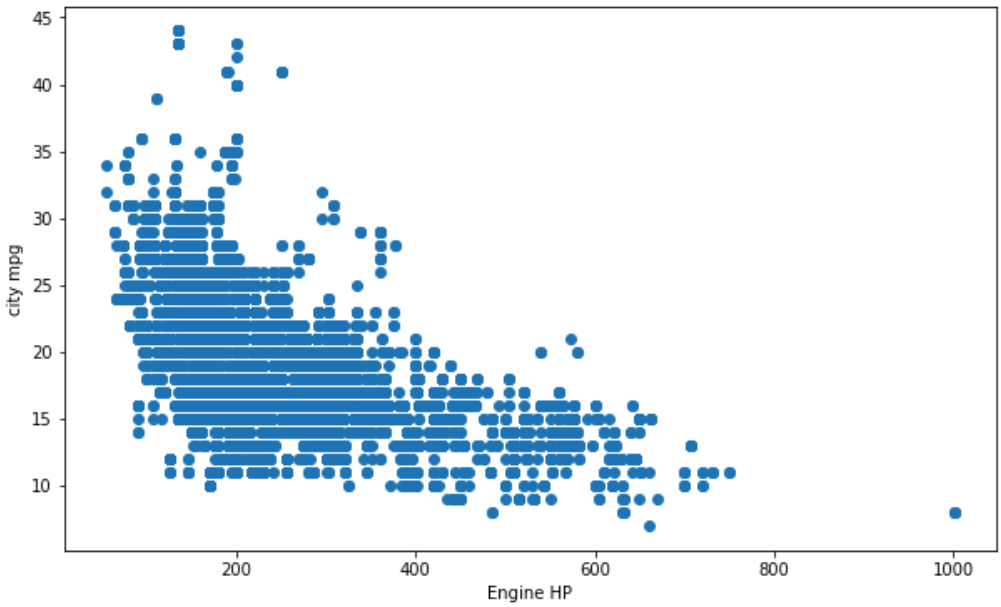
Name: Neeraj Rijhwani Roll No: 64 Class:D15C



Answer: Genesis is least popular while BMW is the most popular brand.

Question 5:

Does higher miles per gallon (MPG) value provide higher horse power?

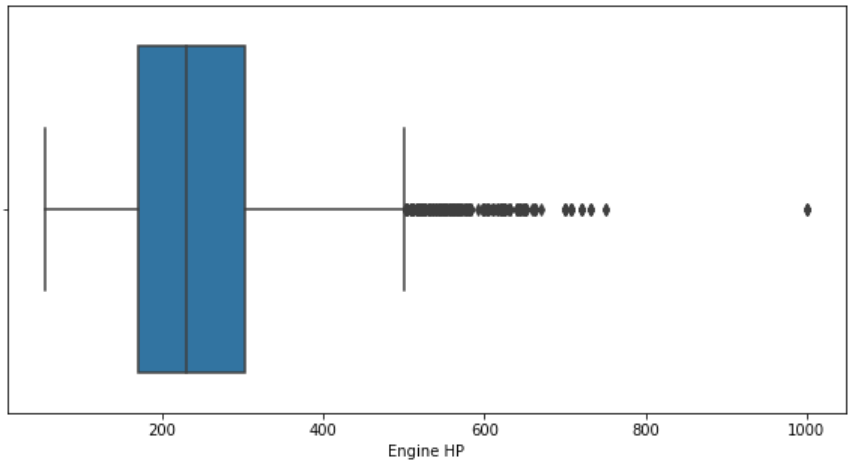


Answer: Scatterplot above suggests that, no, higher MPG value does not provide higher power.

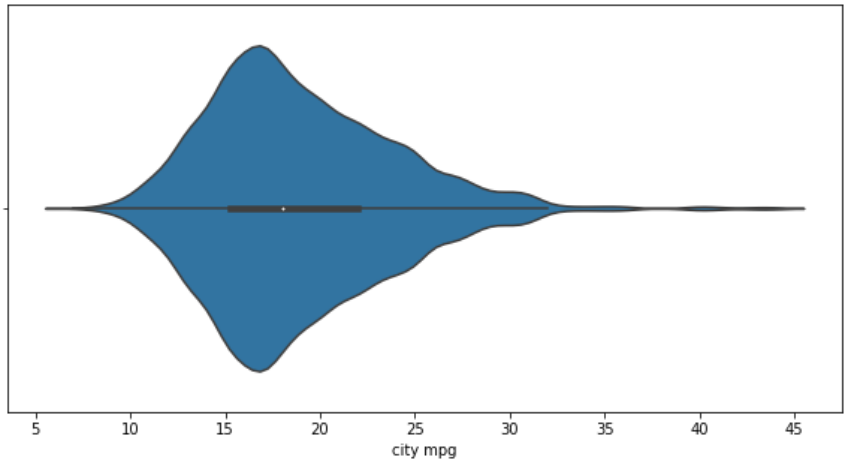
Data Visualizations

Univariate Analysis

```
# BOXPLOT
x = df.select('Engine HP').toPandas()
plt.figure(figsize=(10,5))
sns.boxplot('Engine HP',data=x)
```



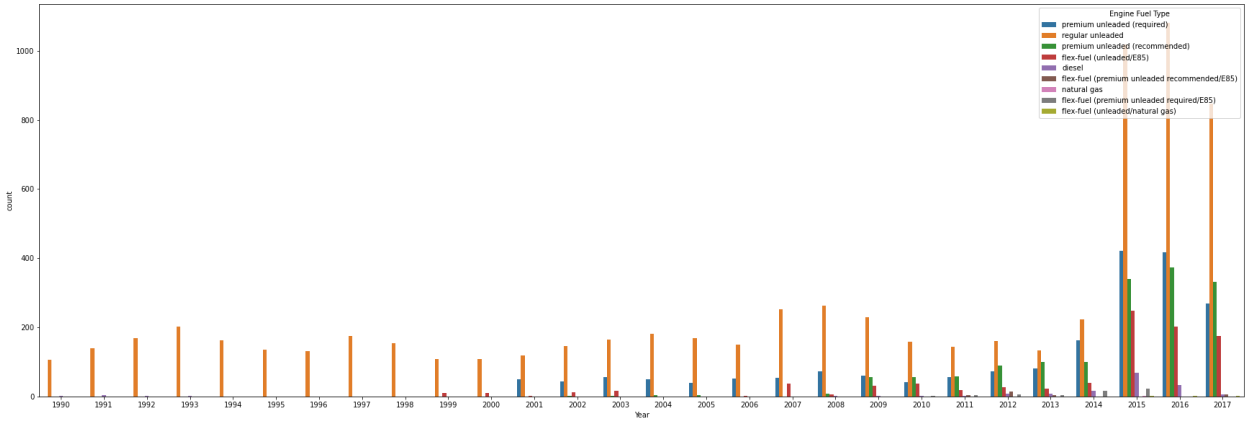

```
# VIOLIN PLOT
x = df.select('city mpg').toPandas()
plt.figure(figsize=(10,5))
sns.violinplot('city mpg',data=x)
```



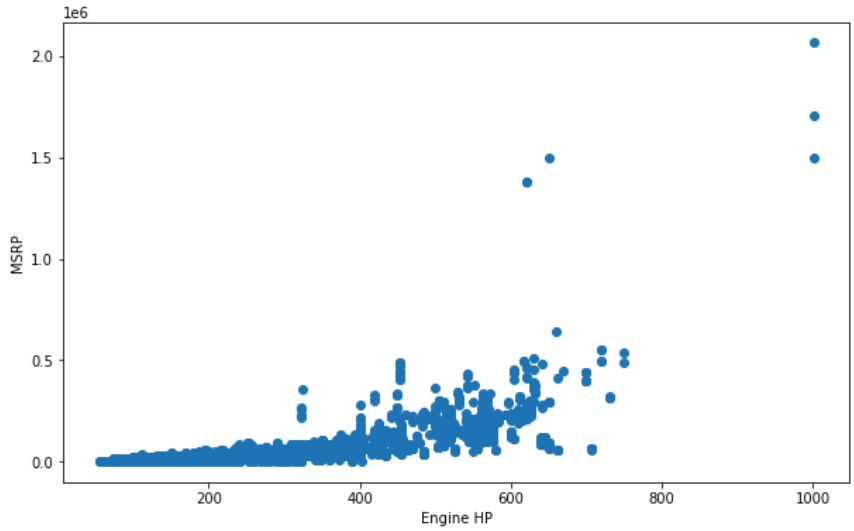
Bivariate Analysis

```
#BAR PLOT
plt.figure(figsize=(30,10))
ax = sns.countplot(x="Year", hue="Engine Fuel Type", data=plot_df)
```

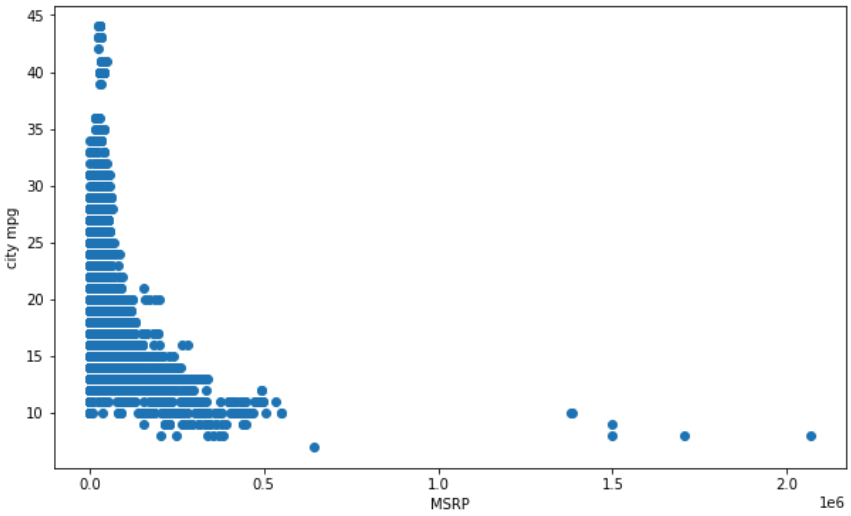
```
plt.show()
```



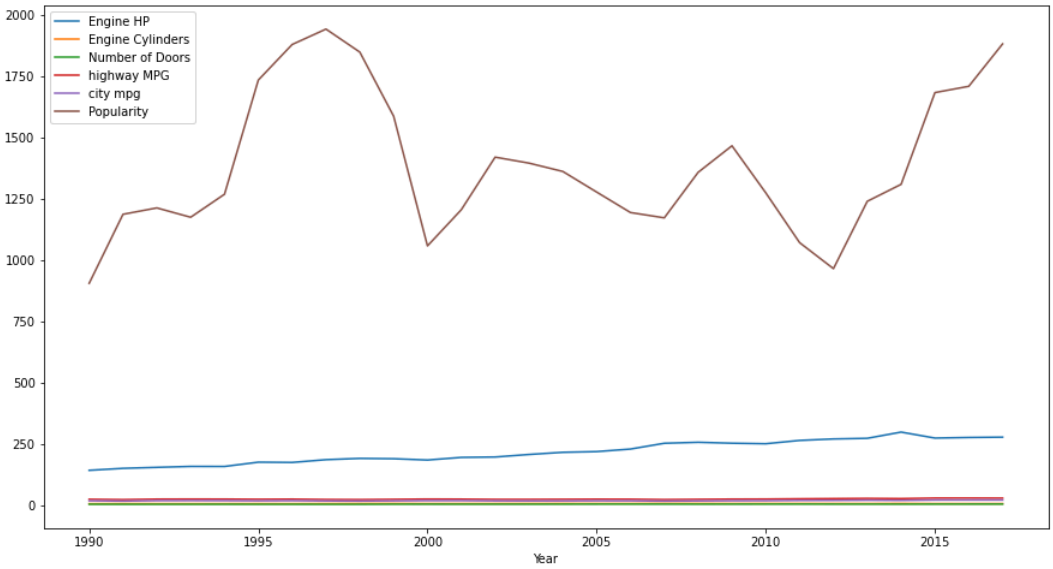

```
#SCATTER PLOT
fig, ax = plt.subplots(figsize=(10,6))
ax.scatter(plot_df['Engine HP'], plot_df['MSRP'])
ax.set_xlabel('Engine HP')
ax.set_ylabel('MSRP')
plt.show()
```



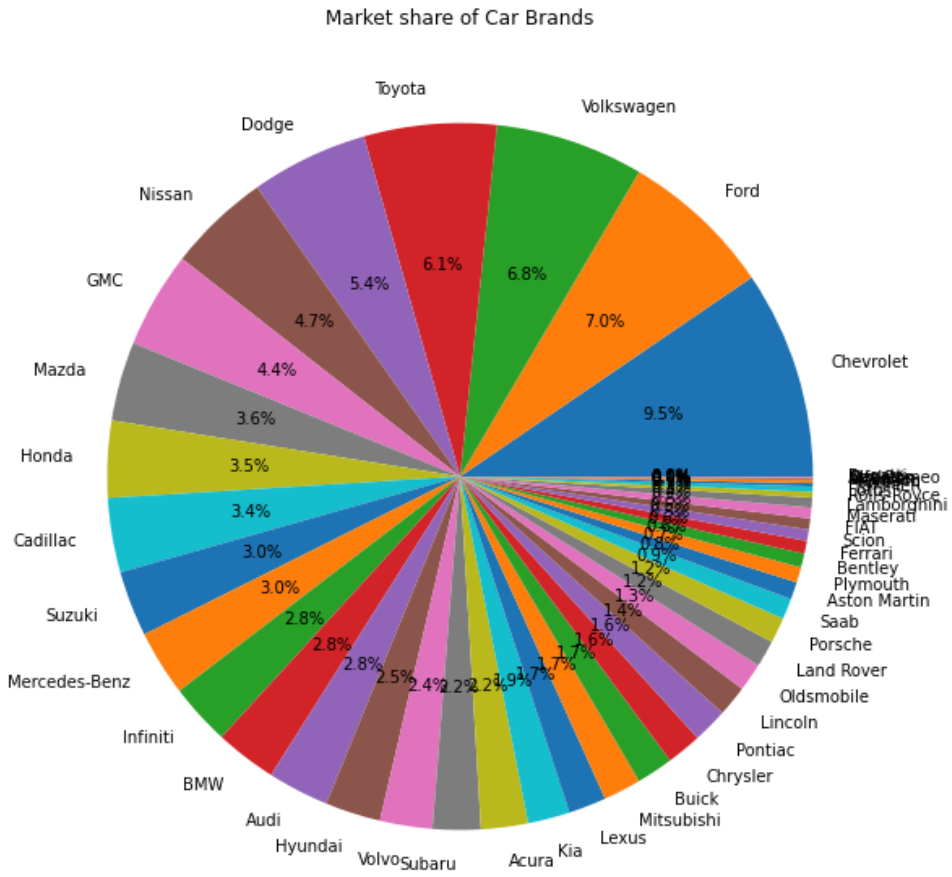

```
#SCATTER PLOT
fig, ax = plt.subplots(figsize=(10,6))
ax.scatter(plot_df['MSRP'], plot_df['city mpg'])
ax.set_xlabel('MSRP')
ax.set_ylabel('city mpg')
plt.show()
```



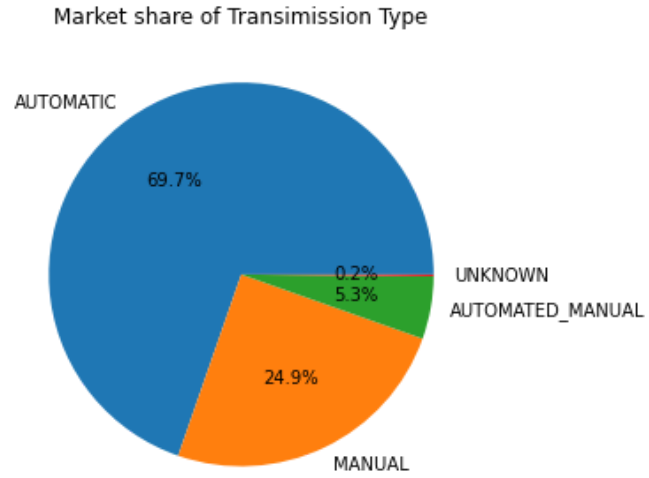

```
#LINE PLOT
df1 = plot_df
df1 = df1.drop(['MSRP'], axis=1)
df1.groupby(['Year']).mean().plot(figsize=(15,8))
```



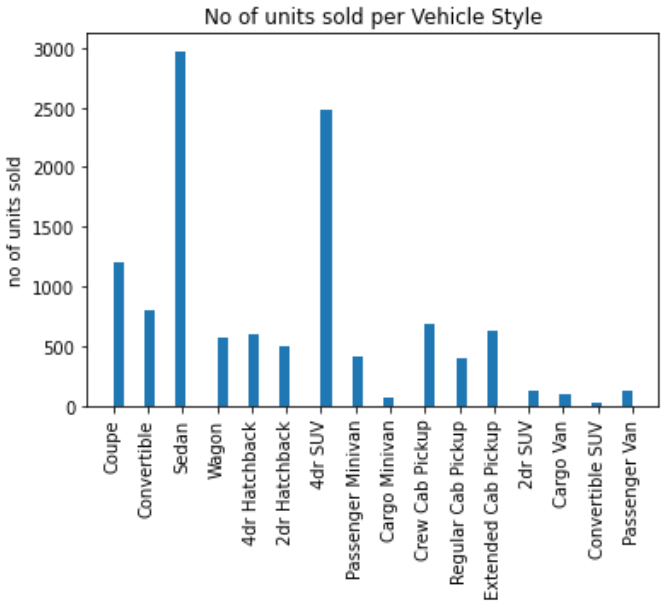
<u>Graphs and some inferences drawn from them</u>



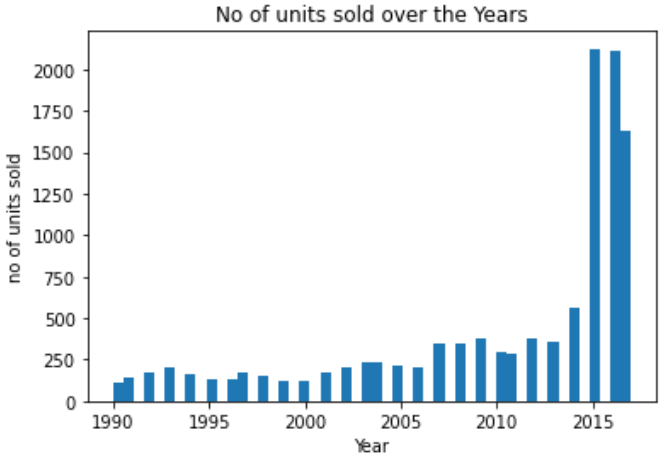
Conclusion: Chevrolet sold the most number of cars



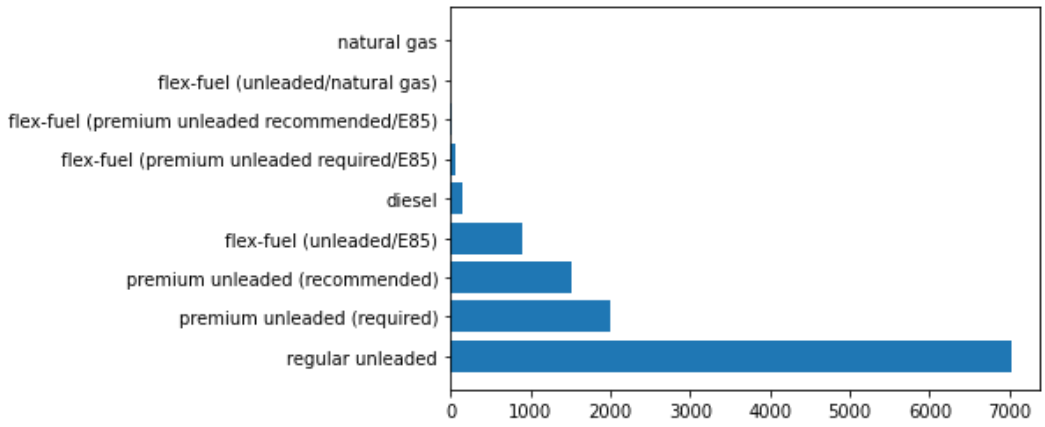
Conclusion: Most people prefer an automatic transmission car over other types



Conclusion: Sedan was the most sold vehicle style.



Conclusion: In the year 2015 there was a spike in car sales



Conclusion: Regular unleaded is the most popular engine fuel type

Conclusion:

Thus, we have learnt what big data is, how Apache Spark is a great big data tool, and also learnt how to use pyspark libraries to preprocess a dataset, and perform EDA on the same in python.