

(7082CEM)

Coursework

Demonstration of a Big Data Program

MODULE LEADER: Dr. Marwan Fuad

Student Name: Saravanacoumar DOURECANNOU

SID: 9806557

Predict car price based on its specification

I can confirm that all work submitted is my own: Yes

Introduction:

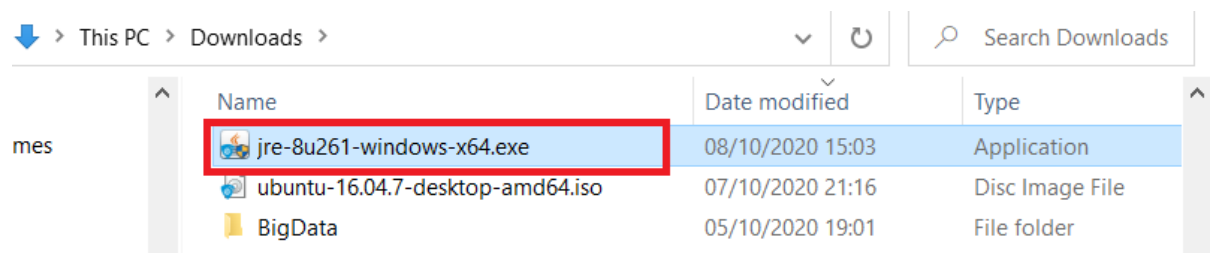
In the recent years, many companies like AutoTrader, Carwow and CarGurus have started estimating the price of the car based on its make, model, engine, wheelbase and so on. This has been an interesting topic not only for car enthusiasts but also for the new/used car buyers. We are going to use the AutoMobile dataset from <https://www.kaggle.com/toramky/automobile-dataset> to apply a regression technique to predict the price based on its specification. Our result would allow us to predict a car's price based on the model we created out of the training data.

Pyspark setup:

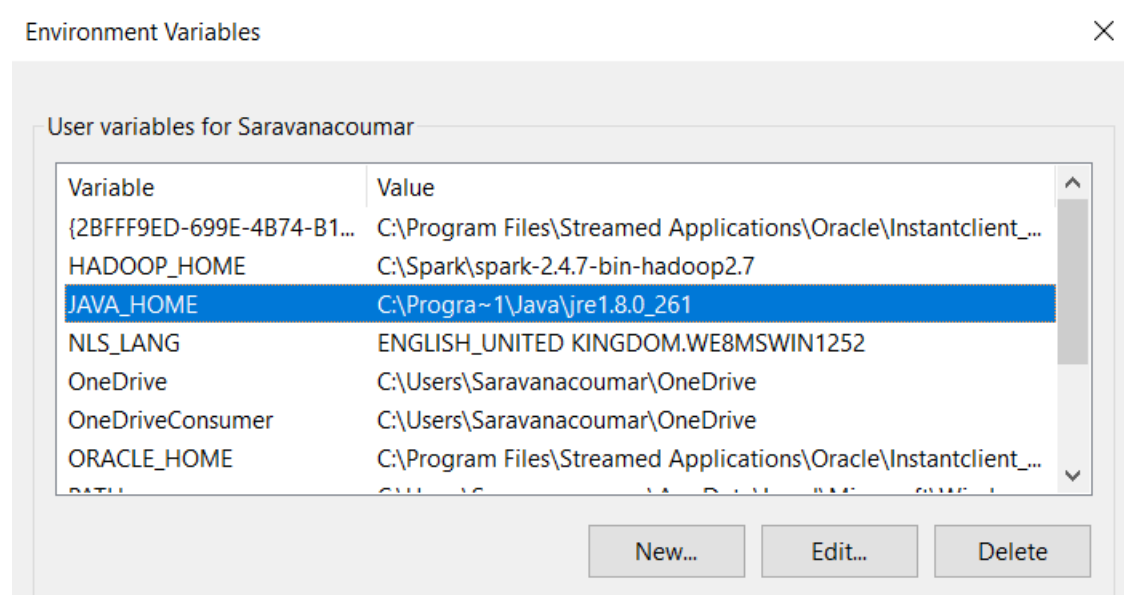
My initial attempt with pyspark setup on Ubuntu went well but I couldn't manage to get Jupyter notebook work with it, Jupyter notebook was my favourite tool for python programming, so I had to redo this work on windows to get it running.

Java:

Java is considered the foundation for Spark framework, so we start our setup by downloading and installing java. I had used java 8 as we had used this version in Hadoop setup on Ubuntu.



After installation, set the system environment variables JAVA_HOME as below:



Now, we can verify if the java is installed successfully by running the following command in the powershell as below:

Windows PowerShell

```
PS C:\Users\Saravanacoumar> java -version
java version "1.8.0_261"
Java(TM) SE Runtime Environment (build 1.8.0_261-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.261-b12, mixed mode)
PS C:\Users\Saravanacoumar>
```

Install Spark:

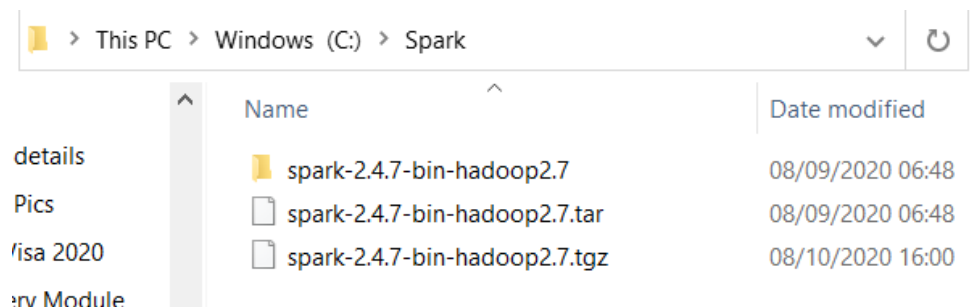
This is an easier step, all we have to do is to just download the Apache spark from <https://spark.apache.org/downloads.html> and select version Spark release 2.4.7 & package type pre-built for Apache Hadoop 2.7, and download spark-2.4.7-bin-hadoop2.7.tgz

Download Apache Spark™

1. Choose a Spark release:
2. Choose a package type:
3. Download Spark: [spark-2.4.7-bin-hadoop2.7.tgz](#)
4. Verify this release using the 2.4.7 [signatures](#), [checksums](#) and [project release KEYS](#).

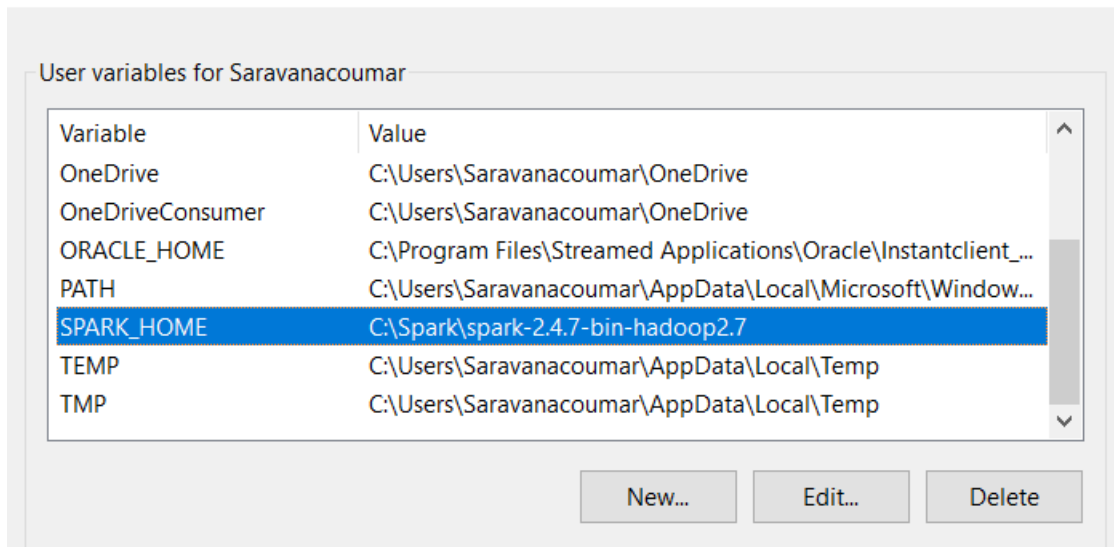
Note that, Spark 2.x is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12. Spark 3.0+ is pre-built with Scala 2.12.

Extract this spark-2.4.7-bin-hadoop2.7.tgz to get the spark-2.4.7-bin-hadoop2.7.tar file, then extract this tar file to a folder spark-2.4.7-bin-hadoop2.7



This tar file contains the whole stuffs that are needed to do the big data stuff, then we set the SPARK_HOME in the system environment variable.

Environment Variables

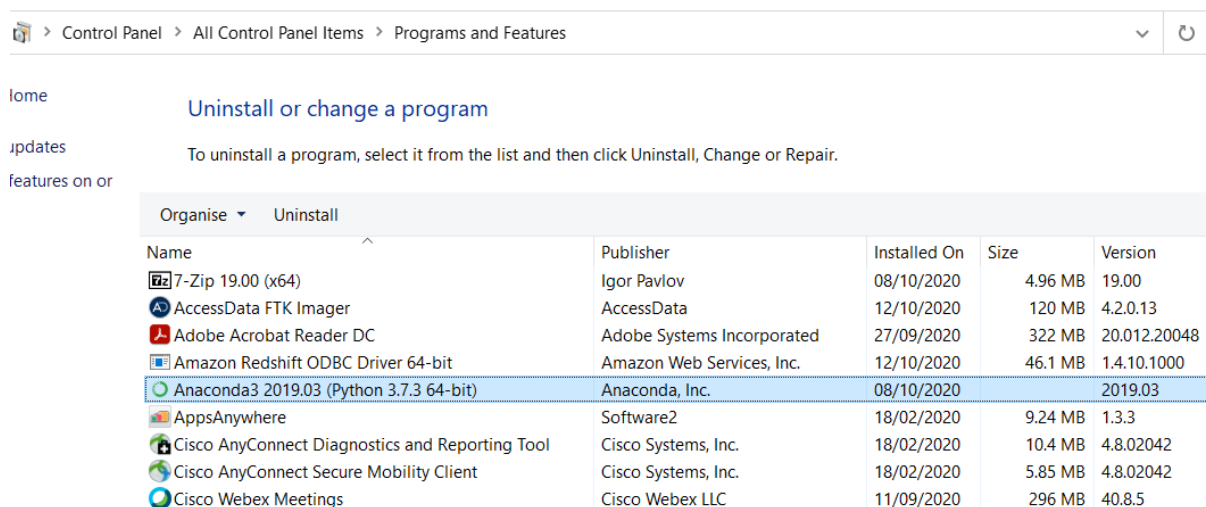


but in order to do pyspark, we need a python interpreter.

The simplest thing to do is to just install python but I needed Jupyter notebook, it is easier if I could install the Anaconda package, which would include python by default.

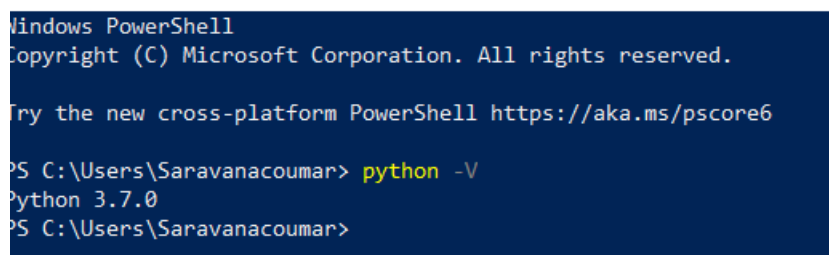
Installing Anaconda:

Download anaconda package and install the individual edition toolkit.




The latest anaconda edition had python 3.9.2 which wasn't supported by pyspark at the moment, following the errors on google led me to downgrade my python version to 3.7.0.

Windows PowerShell



After downgrading my python version, I was successfully able to launch pyspark from powershell



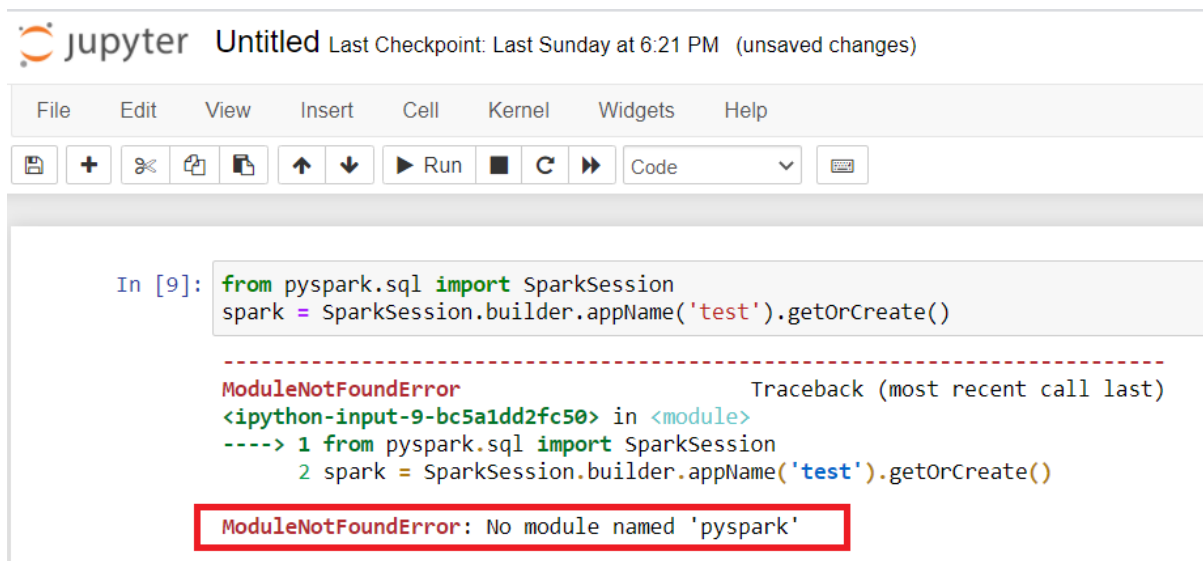
```
PS C:\Users\Saravanacoumar> pyspark
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
20/10/21 20:12:47 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
20/10/21 20:12:51 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Welcome to

  ____      _
 / ___|    / \
| |  | |  / _ \
| |  | | / ___ \
| |  | || |___|
|_|  |_| \____/
version 2.4.7

Using Python version 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018 04:59:51)
SparkSession available as 'spark'.
>>>
```

Now that we have pyspark up and running, its time for setting up jupyter notebook. I used pip command to install jupyterlab and notebook, which allowed me to launch jupyter notebook.

But I was only able to run just python code not pyspark. Jupyter notebook couldn't find the pyspark module.



```
Jupyter Untitled Last Checkpoint: Last Sunday at 6:21 PM (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
[+] [x] [copy] [paste] [undo] [redo] [run] [stop] [refresh] [code]
In [9]: from pyspark.sql import SparkSession
        spark = SparkSession.builder.appName('test').getOrCreate()

-----
ModuleNotFoundError                                Traceback (most recent call last)
<ipython-input-9-bc5a1dd2fc50> in <module>
----> 1 from pyspark.sql import SparkSession
      2 spark = SparkSession.builder.appName('test').getOrCreate()

ModuleNotFoundError: No module named 'pyspark'
```

<https://stackoverflow.com/questions/34302314/no-module-name-pyspark-error/34347373>

had a suggestion to use findspark module in anaconda to not mess with the setup.

```
In [10]: import findspark
findspark.init('C:\Spark\spark-2.4.7-bin-hadoop2.7')

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('test').getOrCreate()
```

```
In [11]: spark
```

```
Out[11]: SparkSession - in-memory
SparkContext
```

[Spark UI](#)

Version

v2.4.7

Master

local[*]

AppName

test

The findspark module did the trick and I was able to create a pyspark session at last.

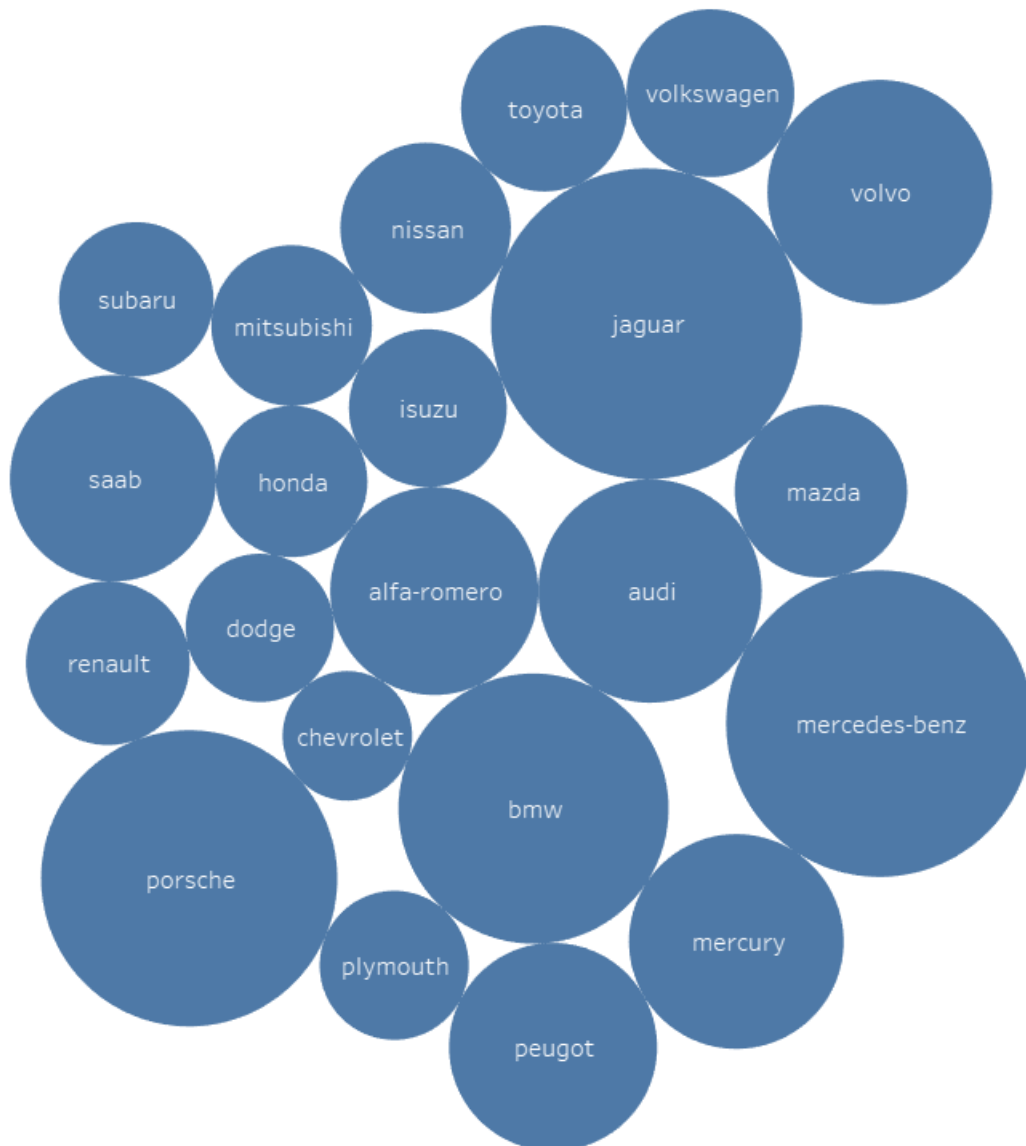
Dataset:

Attribute	Description	Data type
symboling	Range between -3 to +3 (+3 (risky), -3(safe))	Int
normalized-losses	Range between 65 to 256	Int
make	Car manufacturer	String
fuel-type	Gas/Diesel	String
aspiration	Standard or Turbo	String
num-of-doors	2 door / 4 door	String
body-style	Hardtop, wagon, sedan, hatchback, convertible.	String
drive-wheels	rwd, fwd, 4wd	String
engine-location	Rear/Front	String
wheel-base	continuous from 86.6 to 120.9	Double
length	continuous from 141.1 to 208.1	Double
width	continuous from 60.3 to 72.3	Double
height	continuous from 47.8 to 59.8	Double
curb-weight	continuous from 1488 to 4066	Int
engine-type	dohc, dohcv, l, ohc, ohcf, ohcv, rotor	String
num-of-cylinders	eight, five, four, six, three, twelve, two	String
engine-size	continuous from 61 to 326	Int
fuel-system	1bbl, 2bbl, 4bbl, idi, mfi, mpfi, spdi, spfi	String
bore	continuous from 2.54 to 3.94	Double
stroke	continuous from 2.07 to 4.17	Double
compression-ratio	continuous from 7 to 23	Int
horsepower	continuous from 48 to 288	Int
peak-rpm	continuous from 4150 to 6600	Int
city-mpg	continuous from 13 to 49	Int
highway-mpg	continuous from 16 to 54	Int
price	continuous from 5118 to 45400	Int

The first thing we can try out with our data set is to understand the patterns in it but understanding the patterns by looking at an csv file is not easy task or a right thing to do, unless we intend to write lot of code to analyse the statistical bit to get a flavour on it. Another idea would be to use a data analysis tools like Tableau which could give us an insight of the data in a well-represented chart/graph format.

Understanding the average car price by manufacturer:

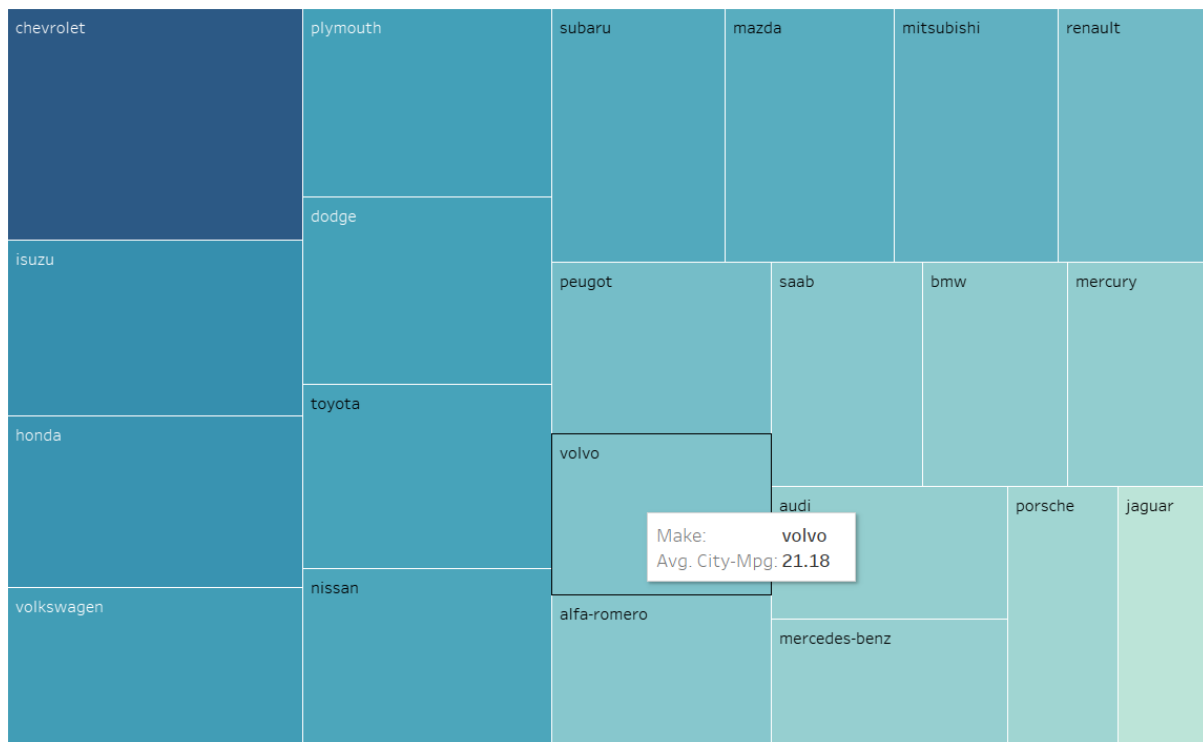
This diagram below certainly gives the overall picture of the price of each manufacturer compared to others, As an example, we can clearly see that on an average Jaguar is the costliest brand in this data set, followed by Benz, Porche and BMW, similarly we see that Chevrolet seems to be the cheapest manufacturer. All these numbers are true to the dataset provided.



Understanding the mileage economy:

Even though Chevrolet makes the cheapest car on average as per the dataset, the cars it makes give the best miles per gallon economy. The costliest manufacturer as per the previous graph is the worst in terms of mileage.

As an indication, all luxury brands struggle with fuel economy when compared to other not so luxury manufacturers.



Engine size to fuel economy mystery:

It is generally assumed that larger the engine size, lesser the fuel economy. This is by the fact that larger engines burn more fuel per combustion cycle. This could be true but the data set for each manufacturer makes my assumption a bit flaky.

The following chart (chart: Engine size Vs city MPG) shows the claim is not entirely true, as an example, we can look at Renault and Saab, we see that Renault with larger engines still do better mpg than Saab with smaller engines.

In all other cases, it holds true that bigger engines lower mpg, smaller engines higher mpg.

Engine location to compression ratio:

The chart (chart: Engine location to compression ratio) below shows that only Porsche does the rear engines and all other manufacturers do front engines, average compression ratio of Benz, Peugeot & VW seems to be higher than that of other car manufacturers. All other manufacturers have very similar compression ratio.

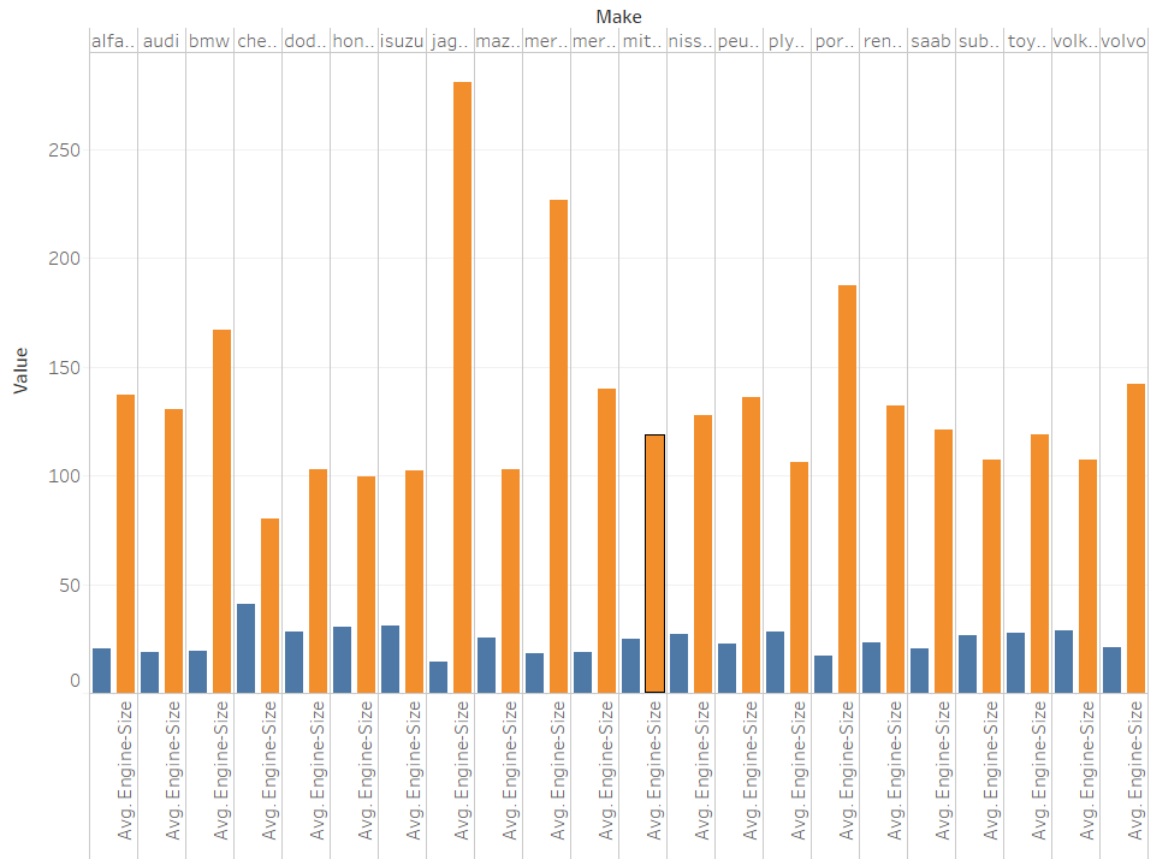


Chart: Engine size Vs city MPG

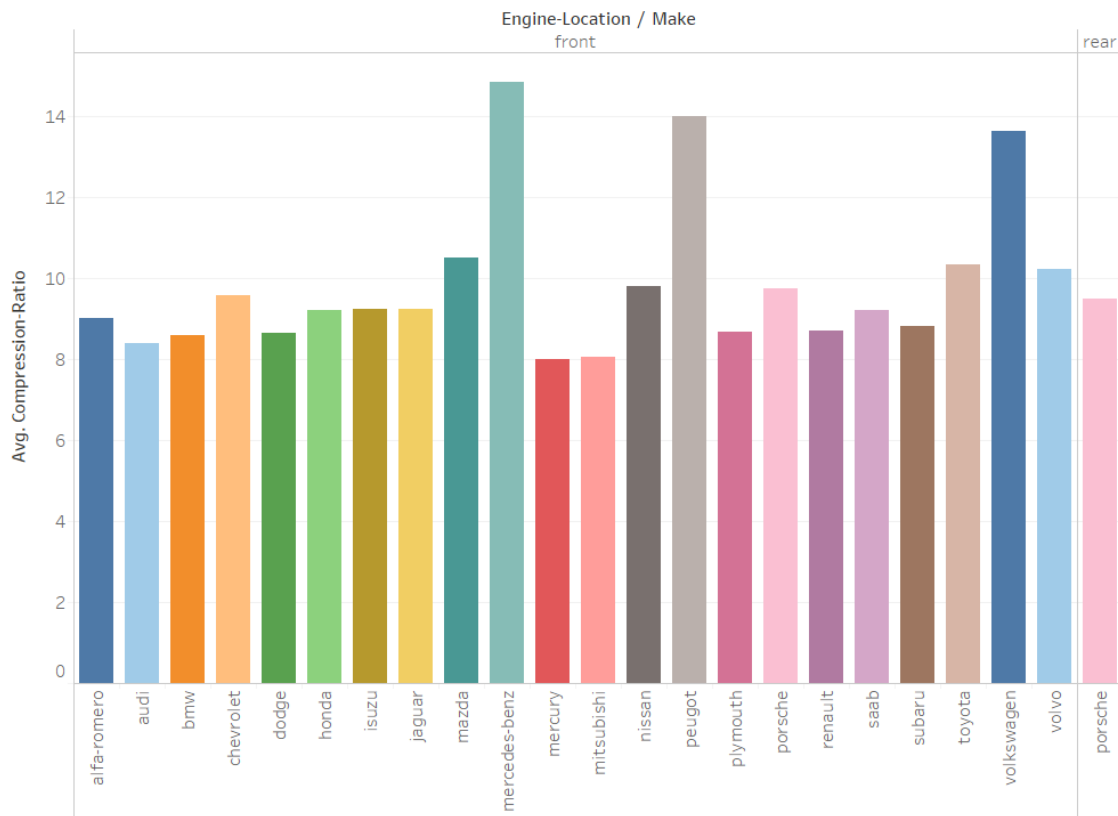
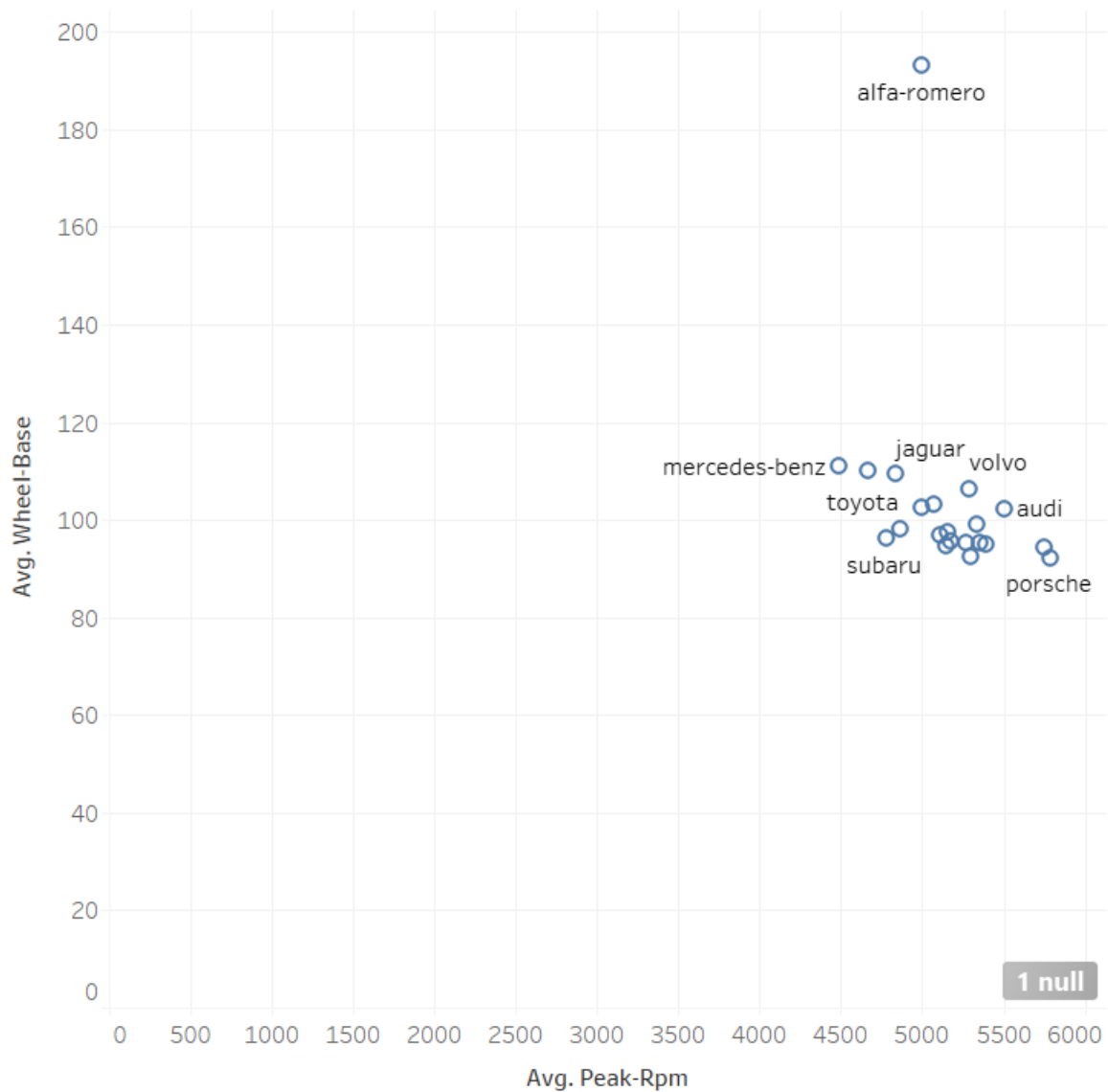


Chart : Engine location to compression ratio

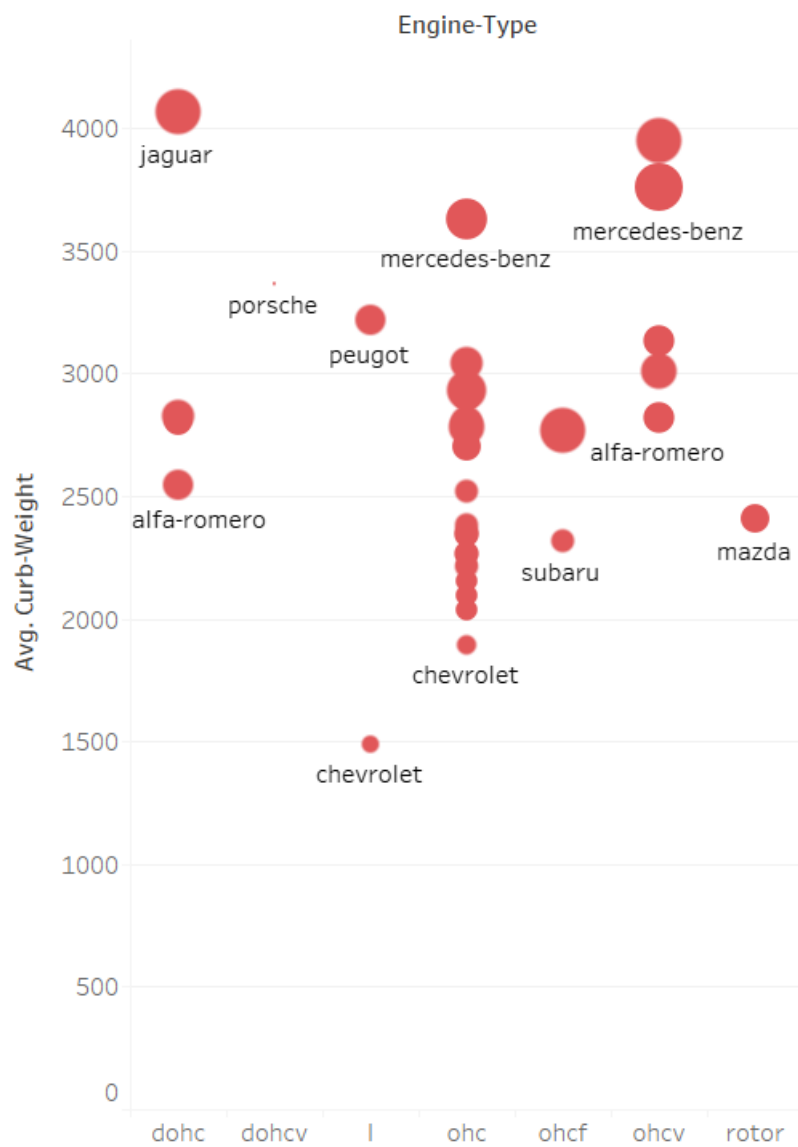
Wheelbase to RPM:

Alfa Romeo appears to be outlier when compared other manufacturers with average wheelbase of more than 180 but still achieving the rpm at 5000.



Curb weight to Engine type:

We are going to plot the curb weight to each engine type, so it is the plot between the manufacturers to engine type to kerb weight, it is more of a 3D graph plotted smartly in 2D like the above one.



Now that we have a better understanding on our data set, we can proceed to apply our machine learning techniques to explore our data further.

Applying Regression:

Reading the Data set:

One of the initial steps to realize when working with Spark is stacking an informational index into a dataframe. If information has been put into a dataframe, we can apply changes, perform calculation and displaying, make representations, and save the outcomes.

The first step is to load the Automobile_data.csv document into a Spark dataframe as demonstrated as follows. This code scrap determines the way of the CSV document and passes various params to the read function to deal with the record. The last part shows the schema of the csv file we just read.

```
In [2]: from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('AutomobileData').getOrCreate()
dataFields = spark.read.csv('C:\\Users\\Saravanacoumar\\Downloads\\BigDataCoursework\\Automobile_data.csv',
                             header = True, inferSchema = True, nullValue='?',
                             ignoreLeadingWhiteSpace=True,
                             ignoreTrailingWhiteSpace=True)

dataFields.printSchema()

root
|-- symboling: integer (nullable = true)
|-- normalized-losses: integer (nullable = true)
|-- make: string (nullable = true)
|-- fuel-type: string (nullable = true)
|-- aspiration: string (nullable = true)
|-- num-of-doors: string (nullable = true)
|-- body-style: string (nullable = true)
|-- drive-wheels: string (nullable = true)
|-- engine-location: string (nullable = true)
|-- wheel-base: double (nullable = true)
|-- length: double (nullable = true)
|-- width: double (nullable = true)
|-- height: double (nullable = true)
|-- curb-weight: integer (nullable = true)
|-- engine-type: string (nullable = true)
|-- num-of-cylinders: string (nullable = true)
|-- engine-size: integer (nullable = true)
|-- fuel-system: string (nullable = true)
|-- bore: double (nullable = true)
|-- stroke: double (nullable = true)
|-- compression-ratio: double (nullable = true)
|-- horsepower: integer (nullable = true)
|-- peak-rpm: integer (nullable = true)
|-- city-mpg: integer (nullable = true)
|-- highway-mpg: integer (nullable = true)
|-- price: integer (nullable = true)
```

Here, setting the inferSchema to true, we ask spark to infer the data types of each column in the csv file and header to true makes the first row of the csv file to be the header of each column.

This particular dataset had the missing values in the form of '?', for some reason, these '?' weren't treated as null values by spark, so we specifically instructed spark that the null values in this data is marked as '?', the remaining trailing white space and leading white space as a precautionary measure just in case the '?' had any white spaces with it.

The read.csv() returns a spark data frame as the result, with which we printed out the schema read from the file.

Spark Dataframe:

The key information type utilized in PySpark is the Spark dataframe. This object is like a table distributed over a cluster and has usefulness that is very similar dataframes in R and Pandas.

It is likewise possible to utilize Pandas dataframes when utilizing Spark, by calling toPandas() on a Spark dataframe, which restores a pandas object. In any case, this capacity ought to by and large be maintained a strategic distance from aside from when working with little dataframes, on the grounds that it pulls the whole item into memory on a single node.

One of the key contrasts among Pandas and Spark dataframes is anxious versus lazy execution. In PySpark, tasks are postponed until an outcome is really required in the pipeline. For instance, you can indicate tasks for stacking an informational index from csv and applying various changes to the dataframe, yet these activities won't promptly be applied. Rather, a graph of changes is recorded, and once the information is really required, for instance when composing the outcomes back to csv, at that point the changes are applied as a solitary pipeline activity. This methodology is utilized to abstain from pulling the full information outline into memory and empowers more successful handling over a group of machines. With Pandas dataframes, everything is stored into memory, and each panda activity is quickly applied.

As a thumb rule, it's a best practice to keep away from eager tasks in Spark if conceivable, since it restricts the amount of your pipeline can be viably distributed.

Sampling the data:

We can look in the data frame read from the csv file as follows by looking at only 5 values in each column:

```
In [4]: from pandas import DataFrame
DataFrame(dataFields.take(5), columns=dataFields.columns).transpose()
```

Out[4]:

	0	1	2	3	4
symboling	3	3	1	2	2
normalized-losses	NaN	NaN	NaN	164	164
make	alfa-romero	alfa-romero	alfa-romero	audi	audi
fuel-type	gas	gas	gas	gas	gas
aspiration	std	std	std	std	std
num-of-doors	2	2	2	4	4
body-style	convertible	convertible	hatchback	sedan	sedan
drive-wheels	rwd	rwd	rwd	fwd	4wd
engine-location	front	front	front	front	front
wheel-base	88.6	88.6	94.5	99.8	99.4
length	168.8	168.8	171.2	176.6	176.6
width	64.1	64.1	65.5	66.2	66.4
height	48.8	48.8	52.4	54.3	54.3
curb-weight	2548	2548	2823	2337	2824
engine-type	dohc	dohc	ohcv	ohc	ohc
num-of-cylinders	four	four	six	four	five
engine-size	130	130	152	109	136
fuel-system	mpfi	mpfi	mpfi	mpfi	mpfi
bore	3.47	3.47	2.68	3.19	3.19
stroke	2.68	2.68	3.47	3.4	3.4
compression-ratio	9	9	9	10	8
horsepower	111	111	154	102	115
peak-rpm	5000	5000	5000	5500	5500
city-mpg	21	21	19	24	18
highway-mpg	27	27	26	30	22
price	13495	16500	16500	13950	17450

Convert String to Integers:

In this dataset, we see that the number of doors and number of cylinders are marked as string like “one”, “two” ...and so on. I think it would make more sense to treat them as Integers as we are going to use only the numerical features in the data set for our regression problem. It is better to have a function that does this.

```
In [3]: from pyspark.sql.types import IntegerType
        from pyspark.sql.functions import udf

        def toNum(noInStr):
            if noInStr == "two":
                return 2
            elif noInStr == "four":
                return 4
            elif noInStr == "five":
                return 5
            elif noInStr == "six":
                return 6
            elif noInStr == "eight":
                return 8
            elif noInStr == "twelve":
                return 12

            return noInStr

        strToNumConvertFunc = udf(toNum, IntegerType())
        dataFields = dataFields.withColumn('num-of-doors',strToNumConvertFunc(dataFields['num-of-doors']))
```

Since the data frame is immutable, we create a new data frame and assign it on the dataFields variable. This changes all the string values such “one” to 1, and “two” to 2 and so on.

The same thing is done for number of cylinders as below:

```
In [9]: dataFields = dataFields.withColumn('num-of-cylinders',strToNumConvertFunc(dataFields['num-of-cylinders']))
        numeric_features = [type[0] for type in dataFields.dtypes if type[1] == 'int' or type[1] == 'double']
        numeric_features

Out[9]: ['symboling',
        'normalized-losses',
        'num-of-doors',
        'wheel-base',
        'length',
        'width',
        'height',
        'curb-weight',
        'num-of-cylinders',
        'engine-size',
        'bore',
        'stroke',
        'compression-ratio',
        'horsepower',
        'peak-rpm',
        'city-mpg',
        'highway-mpg',
        'price']
```

We notice that the columns number of doors and number of cylinders are now considered a number type(Int or Double) such that we can use these features to our regression model.

Filter, select & distinct:

The below line shows the usage of filter API on the data frame which applies the condition where make is either “BMW” or “Audi” and in the resulting rows, we chose only the columns make and fuel-type and the output is chained to distinct that only the distinct entries are made visible.

```
In [5]: dataFields.filter(((dataFields['make'] == 'bmw') | (dataFields['make'] == 'audi'))).select("make", "fuel-type").distinct().show()
```

make	fuel-type
audi	gas
bmw	gas

These chaining operation makes the pyspark more powerful like the pipes in linux where the output of one command, is the input for next command.

GroupBy and Aggregate:

Another interesting thing we can achieve with this data frame is the groupBy, which groups the rows based on the column, which we can further use to Aggregate such as **sum**, **avg**, **max**, **min** and **count**.

This is pretty much same as that of the helper function in SQL query where we can group and apply those aggregate functions. Here is an example

```
In [21]: from pyspark.sql import functions as F
dataFields.groupBy(["make", "fuel-type"]).agg(F.sum("price"), F.max("price"), F.avg("price")).show()
```

make	fuel-type	sum(price)	max(price)	avg(price)
toyota	diesel	26384	10698	8794.666666666666
volkswagen	diesel	39110	13845	9777.5
mazda	diesel	18344	18344	18344.0
toyota	gas	289962	17669	9998.689655172413
plymouth	gas	55744	12764	7963.428571428572
mercury	gas	16503	16503	16503.0
subaru	gas	102495	11694	8541.25
mazda	gas	99880	18280	9080.0
mercedes-benz	gas	155600	45400	38900.0
volvo	gas	176225	22625	17622.5
honda	gas	106401	12945	8184.692307692308
chevrolet	gas	12870	6575	6435.0
alfa-romero	gas	46495	16500	15498.333333333334
audi	gas	107155	23875	17859.166666666668
mercedes-benz	diesel	113576	31600	28394.0
porsche	gas	125602	37028	31400.5
isuzu	gas	17833	11048	8916.5
jaguar	gas	103800	36000	34600.0
bmw	gas	208950	41315	26118.75
mitsubishi	gas	120117	14869	9239.76923076923

only showing top 20 rows

Missing values for normalized-losses:

We see that some of the normalized-losses column have null value in the form of "?", removing these rows would completely remove "Alfa Romeo" and "Isuzu", this is not what we want.

One way to address this issue is to assume a mean value for these makes. This is not ideal but better than leaving these entire car manufacturers


```
In [9]: from pyspark.sql.functions import mean
meanVal=dataFields.select(mean(dataFields['normalized-losses'])).collect()
print('mean value of Sales', meanVal[0][0])
meanSales=int(meanVal[0][0])
meanSales
```

mean value of Sales 122.0

Out[9]: 122

Here, the double 122.0 to converted to 122 to fit the column data type. The resulting data frame is sampled as below:

```
In [10]: dataFields = dataFields.na.fill(meanSales,subset=['normalized-losses'])
dataFields = dataFields.na.drop(how='any')
```

```
In [11]: DataFrame(dataFields.take(10), columns=dataFields.columns).transpose()
```

Out[11]:

	0	1	2	3	4	5	6	7	8	9
symboling	3	3	1	2	2	2	1	1	1	2
normalized-losses	122	122	122	164	164	122	158	122	158	192
make	alfa-romero	alfa-romero	alfa-romero	audi	audi	audi	audi	audi	audi	bmw

Now that we have our data cleaned/pre-processed, we can go ahead and start our regression techniques to understand the data well.

Understanding the cross influence between features:

Given that we have quite a few features, it might be interesting to know if any two features influence each other, it is also called covariance, we can see one influence on another by plotting scatter plot.

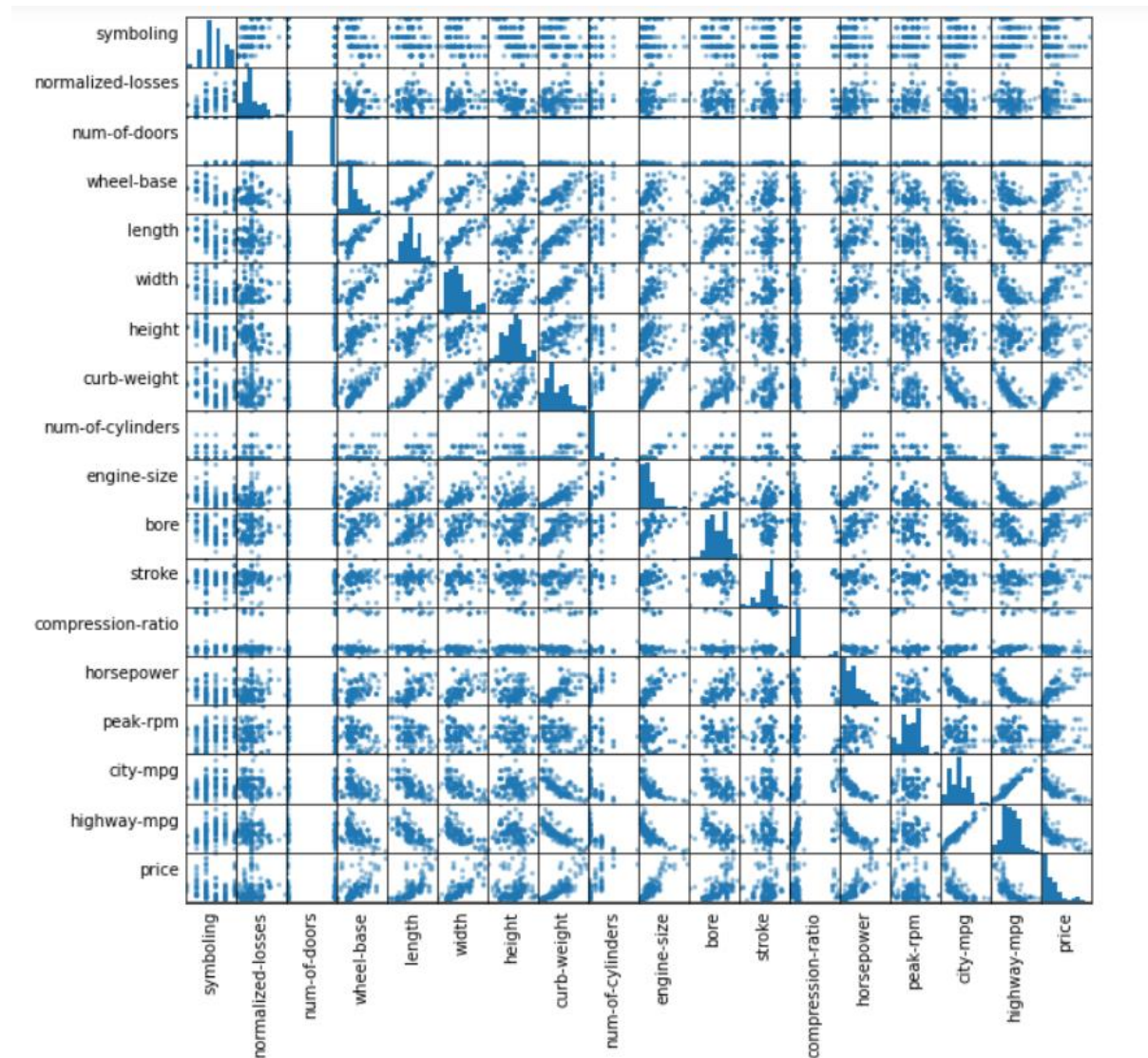
Let's select only the numerical values in the column

```
In [12]: numericData = dataFields.select(numeric_features).toPandas()
```

Scatter plot with all numeric features

```
In [21]: from pandas.plotting import scatter_matrix
import matplotlib
grph = scatter_matrix(numericData, figsize=(10, 10))
noOfColumns = len(numericData.columns)
for i in range(noOfColumns):
    vertico = grph[i, 0]
    vertico.yaxis.label.set_rotation(0)
    vertico.yaxis.label.set_ha('right')
    vertico.set_yticks(())
    horizon = grph[noOfColumns-1, i]
    horizon.xaxis.label.set_rotation(90)
    horizon.set_xticks(())
```

Sadly, we see below that no two features heavily influence the other. In fact, number of doors has absolutely no influence on any other feature.



Vector Assembler:

Using vector assembler, we are going to merge all the numeric features into a vector column, the resulting column is assigned to 'features', since we need to predict price based the other features, we will remove 'price' column from the numeric features list and only consider the other numeric columns for prediction.

The resulting single 'features' column and the label column 'price' are shown as samples below:

```
In [20]: if ('price' in numeric_features):
         numeric_features.remove('price')

         from pyspark.ml.feature import VectorAssembler
         assembler = VectorAssembler(inputCols = numeric_features, outputCol = 'features')
         vDataFields = assembler.transform(dataFields)
         vDataFields = vDataFields.select(['features', 'price'])
         vDataFields.show(3)
```

```
+-----+-----+
|          features|price|
+-----+-----+
|[3.0,122.0,2.0,88...|13495|
|[3.0,122.0,2.0,88...|16500|
|[1.0,122.0,2.0,94...|16500|
+-----+-----+
only showing top 3 rows
```

Train & Test:

After cleaning our data set, we are left with 192 rows down from 206 due to null value columns.

We can now train our algorithm on this data set with 80% of those 192 random rows (143) and apply the prediction on 49(20%) of those remaining rows.

```
In [23]: train, test = vDataFields.randomSplit([0.8, 0.2], seed = 2020)
         print("Training Dataset Count: " + str(train.count()))
         print("Test Dataset Count: " + str(test.count()))
```

```
Training Dataset Count: 143
Test Dataset Count: 49
```

Fit with Linear Regression:

Using the vector assembled numeric features as featuresCol and price as labelCol, we are going to the LinearRegression model with the data split for training above.

```
In [16]: from pyspark.ml.regression import LinearRegression
         lr = LinearRegression(featuresCol = 'features', labelCol = 'price', maxIter=10)
         lrModel = lr.fit(train)
```

```
In [17]: results=lrModel.evaluate(train)

         print('Rsquared Error :', results.r2)
```

```
Rsquared Error : 0.8745744740277719
```

The closer the R squared value to 1, the better the model is, in this case the R squared value of 87.5% percent is a good indication that the line fits the predicted model close enough to be considered for prediction.

Prediction:

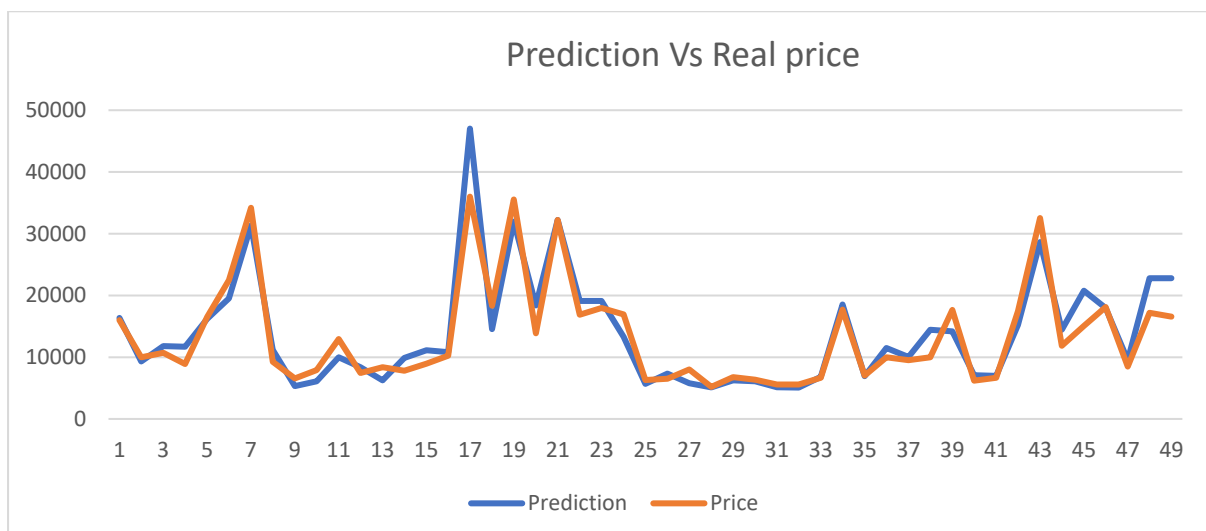
Let's use the model we got from fitting the training set to predict the price on the test set.

```
In [29]: unlabeledData=test.select('features')
predictions=lrModel.transform(test)
predictions.select("prediction", "price").show()

predictions.select("prediction", "price").coalesce(1).write.csv("myresults.csv")
```

```
+-----+-----+
|      prediction|price|
+-----+-----+
| 16373.62280807476|15985|
| 9309.621778286535| 9988|
|11814.402317457883|10698|
|11688.010433060648| 8921|
|16231.282563325782|16515|
|19519.409981871016|22470|
| 31348.71765950633|34184|
|11166.355624216427| 9279|
| 5308.79599750269| 6575|
|6114.4537093758045| 7898|
| 9989.037142010522|12945|
| 8389.7356578299| 7463|
| 6259.666698957073| 8358|
| 9866.852828341362| 7788|
|11150.88504285019| 8949|
|10831.651753398539|10245|
| 47020.21811677068|36000|
|14571.094004244667|18344|
|31950.837489260455|35550|
| 18408.0658606765|13860|
+-----+-----+
only showing top 20 rows
```

The predicted price is surprisingly closer to the real price of the car, so we can conclude that we can use this model to predict a car price up to 87.5% accuracy.



We plotted this graph based on the coalesced csv file and it appears that the prediction and label goes together most of the time.

Conclusion:

Although the data is not very extensive in terms of models, the regression technique applied would be same, and the scalability would also be same as Hadoop grows horizontally.

We were able to predict the price close enough (87.5%) only based on the numeric features, the uncertainty in the prediction could be due to not using the other String features like 'make', 'fuel-type' and 'body-type' etc. In short, we can say that more the features and more the training data, the better the results become.

References:

Mishra, R., n.d. *Pyspark Recipes*.

Luu, H., n.d. *Beginning Apache Spark 2*.

Lee, D. and Drabas, T., n.d. *Pyspark Cookbook*.

Medium. 2020. *Installing Apache Pyspark On Windows 10*. [online] Available at: <<https://towardsdatascience.com/installing-apache-pyspark-on-windows-10-f5f0c506bea1>> [Accessed 21 October 2020].

2020. [online] Available at: <https://www.youtube.com/watch?v=aHaOlvR00So&t=2158s&ab_channel=edureka%21> [Accessed 21 October 2020].

Appendix:

All the coursework related documents including working code is pushed into below git link.

<https://github.com/Saran-san/BigDataCourseWork>