Predict car price based on its specification.

Introduction:

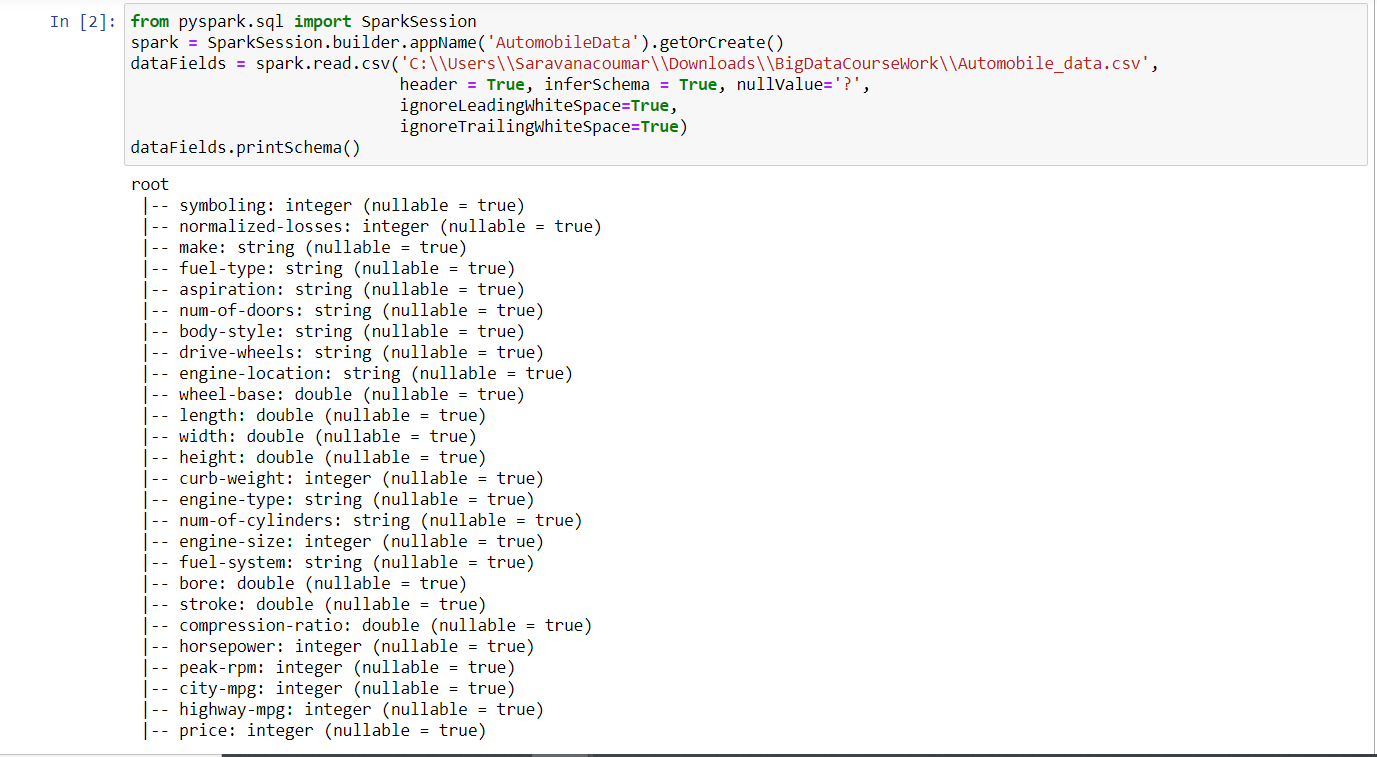
In the recent years, many companies have started estimating the price of the car based on its make, model, engine, wheelbase and so on. This has been an interesting topic not only for car enthusiasts but also for the new/used car buyers. We have a data set which has these following features

|  |  |  |
| --- | --- | --- |
| **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 |

Reading the Data set:

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

The first step is to peruse 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.



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.

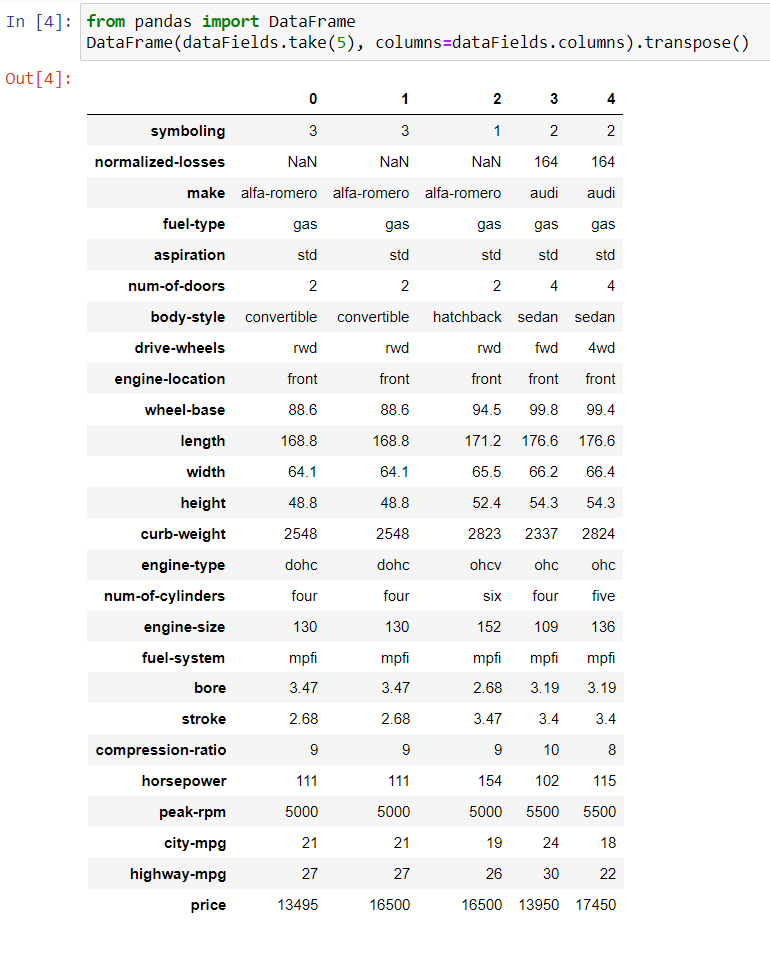
It is likewise possible to utilize Pandas dataframes when utilizing Spark, by calling toPandas() on a Spark dataframe, which restores a pandas object. In any case, this capacity ought to by and large be maintained a strategic distance from aside from when working with little dataframes, on the grounds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



Convert String to Integers:

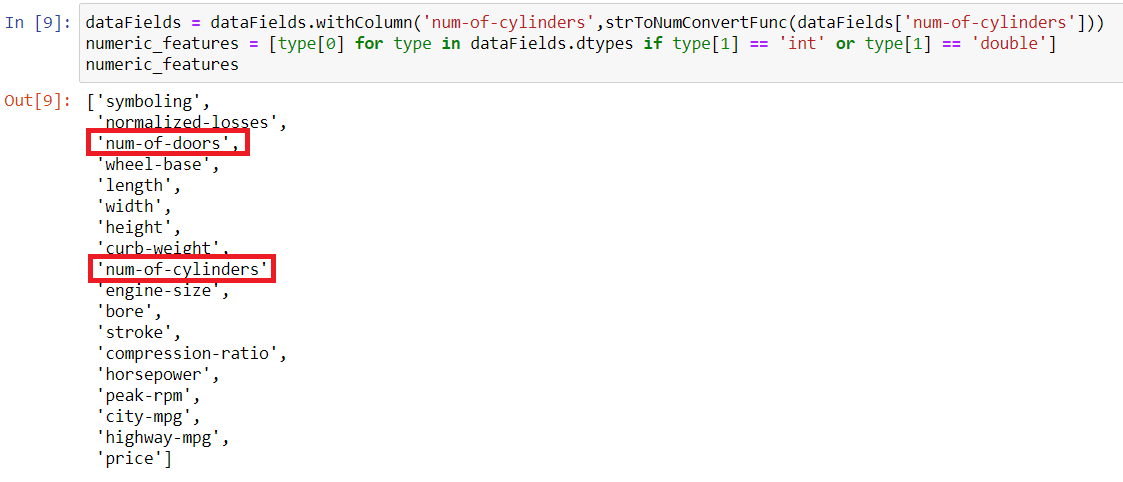
In this dataset, we see that the number of doors and number of cylinders are marked as string like

“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.



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.

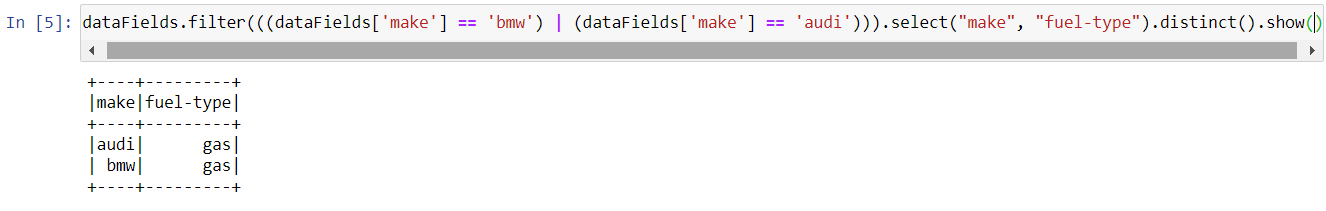
The same thing is done for number of cylinders as below:



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.

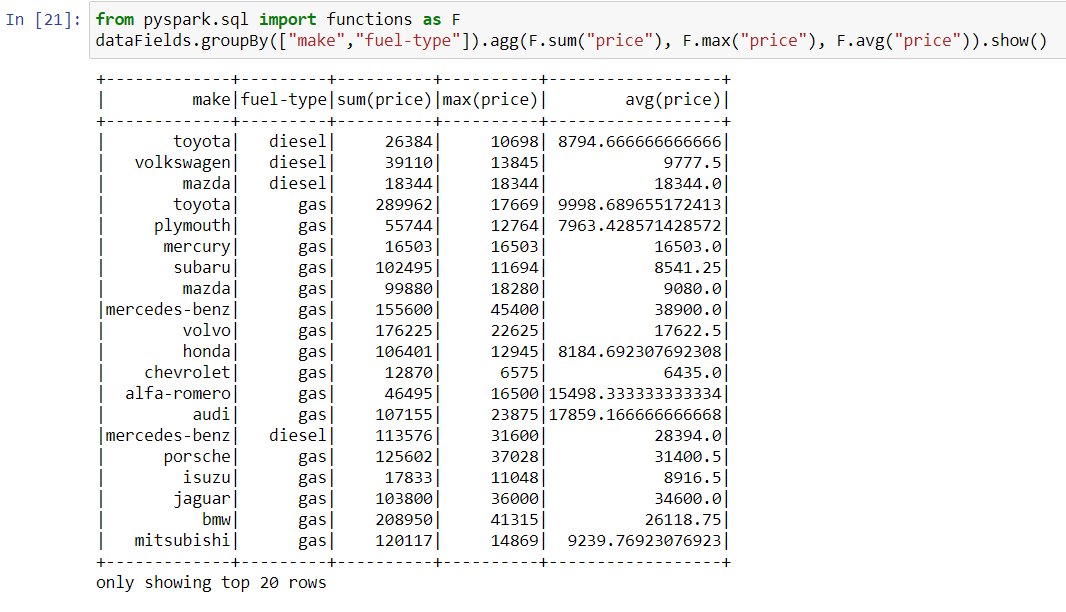


These chaining operation makes the pyspark more powerful like the pipes in linux where the output of one command, is the input for next command.

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.**

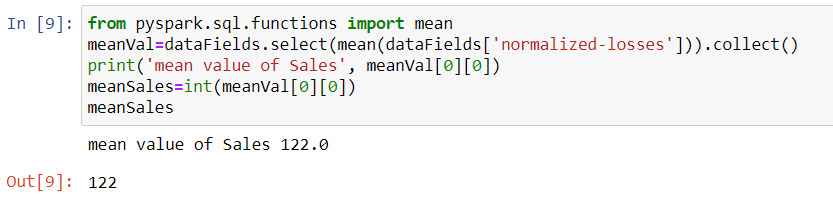
This is pretty much same as that of the helper function in SQL query where we can group and apply those aggregate functions. Here is an example



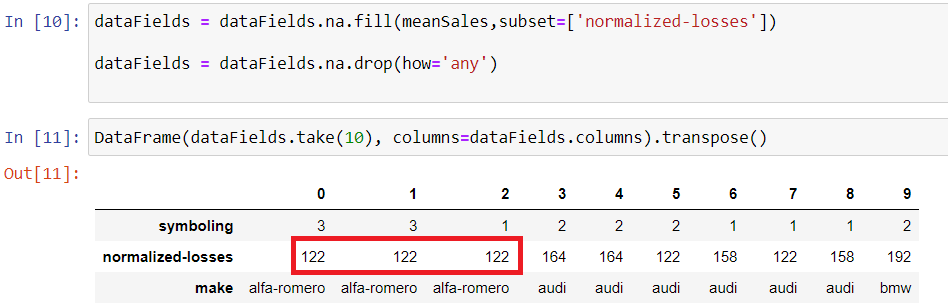
Missing values for normalized-losses:

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



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



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

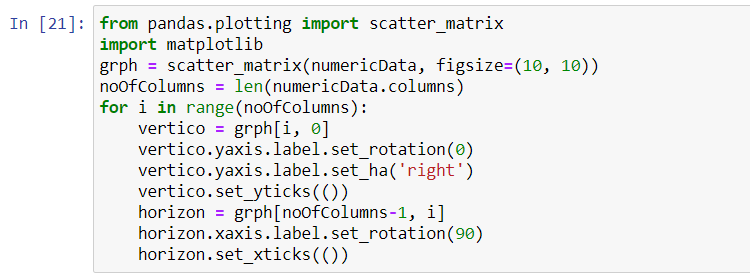
Understanding the cross influence between features:

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



Scatter plot with all numeric features



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.

