**Problem Description:**

An automobile company XYZ from Japan aspires to enter the US market by setting up their manufacturing unit there and producing cars locally to give competition to their US and European counterparts.

They want to understand the factors affecting the pricing of cars in the American market, since those may be very different from the Japanese market. Essentially, the company wants to know:

* Which variables are significant in predicting the price of a car
* How well do those variables describe the price of a car

Based on various market surveys, the consulting firm has gathered a large dataset of different types of cars across the American market.

**Business Objectives:**

You as a Data scientist are required to apply some data science techniques for the price of cars with the available independent variables. That should help the management to understand how exactly the prices vary with the independent variables. They can accordingly manipulate the design of the cars, the business strategy, etc. to meet certain price levels.

The solution is divided into the following sections:

* Data understanding and exploration
* Data cleaning
* Data preparation
* Model building and evaluation

**Data understanding and exploration:**

Summary of data: 205 rows, 26 columns, no null values

The column“**Price**” is the target variable and rest of the columns are independent variables.

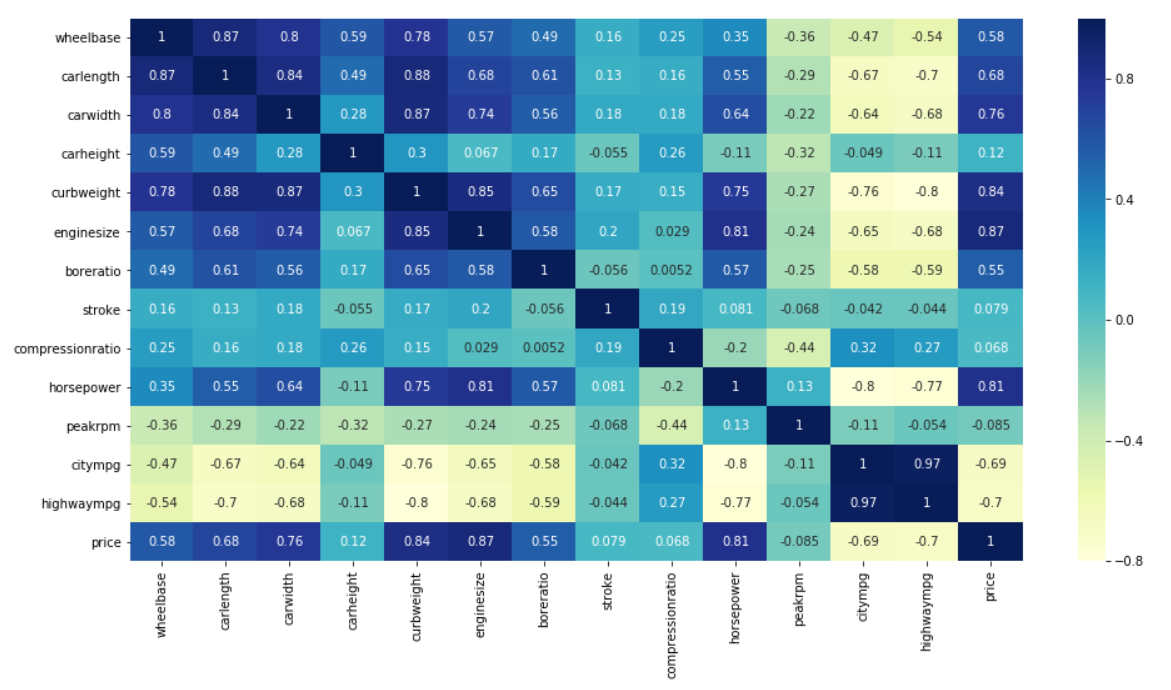
The independent variables are again divided into Categorical and Numerical variables.

**Numerical variables:** [‘wheelbase’, ‘carlength’, ‘carwidth’, ‘carheight’, ‘curbweight’, ‘enginesize’, ‘boreratio’, ‘stroke’, ‘compression ratio’, ‘horsepower’, ‘peak rpm’, ‘city mpg’, ‘highway mpg’]

**Categorical variables:**[‘symboling’, ‘fuel type’, ‘aspiration’, ‘doornumber’, ‘carbody’, ‘drivewheel’, ‘enginelocation’, ‘enginetype’, ‘cylindernumber’, ‘fuelsystem’ ‘car\_name’]

**Heatmap to show correlation of Numerical and Target variable:**

Now let’s plot Heatmap which is pretty useful to visualise multiple correlations among numerical variables. We have also used the Target variable “**Price**” to understand the correlation of numerical variables with it.



The heatmap shows some useful insights:

Correlation of **target variable “Price”** with **independent variables**:

* Price is highly (positively) correlated with wheelbase, carlength, carwidth, curbweight, enginesize, horsepower (notice how all of these variables represent the size/weight/engine power of the car)
* Price is negatively correlated to ‘citympg’ and ‘highwaympg’ (-0.70 approximately). This suggest that cars having high mileage may fall in the ‘economy’ cars category, and are priced lower (think Maruti Alto/Swift type of cars, which are designed to be affordable by the middle class, who value mileage more than horsepower/size of car etc.)

Correlation among independent variables:

* Many independent variables are highly correlated (look at the top-left part of matrix): wheelbase, carlength, curbweight, enginesize etc. are all measures of ‘size/weight’, and are positively correlated

Thus, while building the model, we’ll have to pay attention to multicollinearity (especially linear models, such as linear and logistic regression, suffer more from multicollinearity).

**Data Cleaning:**

We’ve seen that there are no missing values in the dataset.

We’ve also seen that variables are in the correct format, except “symboling”, which should rather be a categorical variable (so that dummy variable are created for the categories).

We have also done data preprocessing on The variable “CarName” and created a new variable called as “car\_company”.

**Data Preparation:**

Let’s now prepare the data for model building.

Split the data into X and y.

X = cars.loc[:, ['symboling', 'fueltype', 'aspiration', 'doornumber','carbody', 'drivewheel', 'enginelocation', 'wheelbase', 'carlength','carwidth', 'carheight', 'curbweight', 'enginetype', 'cylindernumber','enginesize', 'fuelsystem', 'boreratio', 'stroke', 'compressionratio','horsepower', 'peakrpm', 'citympg', 'highwaympg',  
'car\_company']]

y = cars['price']

Creating dummy variables for categorical variables.

# subset all categorical variables  
cars\_categorical = X.select\_dtypes(include=['object'])  
# convert into dummies  
cars\_dummies = pd.get\_dummies(cars\_categorical, drop\_first=True)  
# drop categorical variables   
X = X.drop(list(cars\_categorical.columns), axis=1)  
# concat dummy variables with X  
X = pd.concat([X, cars\_dummies], axis=1)

Scaling the features and getting the final list of columns in dataframe for model building.

# scaling the features  
from sklearn.preprocessing import scale# storing column names in cols, since column names are (annoyingly) lost after   
# scaling (the df is converted to a numpy array)  
cols = X.columns  
X = pd.DataFrame(scale(X))  
X.columns = cols  
X.columns

Text, letter

Description automatically generated

final list of columns in dataframe for model building

Final Train-Test split of data.

# split into train and test  
from sklearn.cross\_validation import train\_test\_split  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,   
 train\_size=0.7,  
 test\_size = 0.3, random\_state=100)

**Model Building and Evaluation:**

Building the first model with all the features

# instantiate  
lm = LinearRegression()# fit  
lm.fit(X\_train, y\_train)  
# predict   
y\_pred = lm.predict(X\_test)# metrics  
from sklearn.metrics import r2\_scoreprint(r2\_score(y\_true=y\_test, y\_pred=y\_pred))

R-squared = 0.83826213934

Not bad, we are getting approx. 83% r-squared with all the variables. Let’s see how much we can get with lesser features.

Let’s now build a model using recursive feature elimination to select features. We’ll first start off with an arbitrary number of features, and then use the “**statsmodels”** library to build models using the shortlisted features (this is because sklearn doesn’t have adjusted r-squared but statsmodels has).

**Choosing the optimal number of features for Model building:**

One way to choose the optimal number of features is to make a plot between number of features(n\_features) vs adjusted r-squared, and then choose the best value of n\_features.

n\_features\_list = list(range(4, 20))  
adjusted\_r2 = []  
r2 = []  
test\_r2 = []for n\_features in range(4, 20):# RFE with n features  
 lm = LinearRegression()# specify number of features  
 rfe\_n = RFE(lm, n\_features)# fit with n features  
 rfe\_n.fit(X\_train, y\_train)# subset the features selected by rfe\_6  
 col\_n = X\_train.columns[rfe\_n.support\_]# subsetting training data for 6 selected columns  
 X\_train\_rfe\_n = X\_train[col\_n]# add a constant to the model  
 X\_train\_rfe\_n = sm.add\_constant(X\_train\_rfe\_n)# fitting the model with 6 variables  
 lm\_n = sm.OLS(y\_train, X\_train\_rfe\_n).fit()  
 adjusted\_r2.append(lm\_n.rsquared\_adj)  
 r2.append(lm\_n.rsquared)  
   
   
 # making predictions using rfe\_15 sm model  
 X\_test\_rfe\_n = X\_test[col\_n]# # Adding a constant variable   
 X\_test\_rfe\_n = sm.add\_constant(X\_test\_rfe\_n, has\_constant='add')# # Making predictions  
 y\_pred = lm\_n.predict(X\_test\_rfe\_n)  
   
 test\_r2.append(r2\_score(y\_test, y\_pred))# plotting adjusted\_r2 against n\_features  
plt.figure(figsize=(10, 8))  
plt.plot(n\_features\_list, adjusted\_r2, label="adjusted\_r2")  
plt.plot(n\_features\_list, r2, label="train\_r2")  
plt.plot(n\_features\_list, test\_r2, label="test\_r2")  
plt.legend(loc='upper left')  
plt.show()

Chart, line chart

Description automatically generated

Number of Features vs R-squared

Based on the plot, we can choose the number of features considering the r2\_score we are looking for. Note that there are a few caveats in this approach, and there are more sophisticated techniques to choose the optimal number of features:

* Cross-validation: In this case, we have considered only one train-test split of the dataset; the values of r-squared and adjusted r-squared will vary with train-test split. Thus, cross-validation is a more commonly used technique (you divide the data into multiple train-test splits into ‘folds’, and then compute average metrics such as r-squared across the ‘folds’.
* The values of r-squared and adjusted r-squared are computed based on the training set, though we must always look at metrics computed on the test set. For e.g. in this case, the test r2 actually goes down with increasing n — this phenomenon is called ‘overfitting’, where the performance on training set is good because the model has in some way ‘memorised’ the dataset, and thus the performance on test set is worse.

Thus, we can choose anything between 4 and 12 features, since beyond 12, the test r2 goes down; and at lesser than 4, the r2\_score is too less.

In fact, the test\_r2 score doesn’t increase much anyway from n=6 to n=12. It is thus wiser to choose a simpler model, and so let’s choose n=6.

**Final Model:**

Let’s now build the final model with 6 features.

# RFE with n features  
lm = LinearRegression()n\_features = 6# specify number of features  
rfe\_n = RFE(lm, n\_features)# fit with n features  
rfe\_n.fit(X\_train, y\_train)# subset the features selected by rfe\_6  
col\_n = X\_train.columns[rfe\_n.support\_]# subsetting training data for 6 selected columns  
X\_train\_rfe\_n = X\_train[col\_n]# add a constant to the model  
X\_train\_rfe\_n = sm.add\_constant(X\_train\_rfe\_n)# fitting the model with 6 variables  
lm\_n = sm.OLS(y\_train, X\_train\_rfe\_n).fit()  
adjusted\_r2.append(lm\_n.rsquared\_adj)  
r2.append(lm\_n.rsquared)# making predictions using rfe\_15 sm model  
X\_test\_rfe\_n = X\_test[col\_n]# # Adding a constant variable   
X\_test\_rfe\_n = sm.add\_constant(X\_test\_rfe\_n, has\_constant='add')# # Making predictions  
y\_pred = lm\_n.predict(X\_test\_rfe\_n)test\_r2.append(r2\_score(y\_test, y\_pred))# summary  
lm\_n.summary()

Graphical user interface

Description automatically generatedOLS Regression result

# results   
r2\_score(y\_test, y\_pred)

r2-squared = 0.88514228773125714

So the model has accuracy of 88.51% on test data which is good. There are other ways of model evaluation as well, let’s see those points.

**Final Model Evaluation:**

Let’s now evaluate the model in terms of its assumptions. We should test that:

* The error terms are normally distributed with mean approximately 0.
* There is little correlation between the predictors.
* Homoscedasticity, i.e. the ‘spread’ or ‘variance’ of the error term (y\_true-y\_pred) is constant.

# Error terms  
c = [i for i in range(len(y\_pred))]  
fig = plt.figure()  
plt.plot(c,y\_test-y\_pred, color="blue", linewidth=2.5, linestyle="-")  
fig.suptitle('Error Terms', fontsize=20) # Plot heading   
plt.xlabel('Index', fontsize=18) # X-label  
plt.ylabel('ytest-ypred', fontsize=16) # Y-label  
plt.show()

Chart, histogram

Description automatically generated

Error Terms

Plotting the error terms to understand the distribution.

fig = plt.figure()  
sns.distplot((y\_test-y\_pred),bins=50)  
fig.suptitle('Error Terms', fontsize=20) # Plot heading   
plt.xlabel('y\_test-y\_pred', fontsize=18) # X-label  
plt.ylabel('Index', fontsize=16) # Y-label  
plt.show()

Chart, histogram

Description automatically generatedError distribution

Now it may look like the mean is not 0, though compared to the scale of ‘Price’, -380 is not such a big number (see distribution below).

Chart, histogram

Description automatically generated

Price Distribution

Multicollinearity:

predictors = ['carwidth', 'curbweight', 'enginesize',   
 'enginelocation\_rear', 'car\_company\_bmw', 'car\_company\_porsche']cors = X.loc[:, list(predictors)].corr()  
sns.heatmap(cors, annot=True)  
plt.show()

Graphical user interface

Description automatically generated

Heatmap to show multicollinearity among Predictors

**Conclusion:**

Though this is the most simple model we’ve built till now, the final predictors still seem to have high correlations. One can go ahead and remove some of these features, though that will affect the adjusted-r2 score significantly (you should try doing that).

Thus, for now, the final model consists of the 6 variables mentioned above.