Data Science II with python (Class notes)

STAT 303-2-Sec20&21

2025-01-07

Table of contents

Preface

These are class notes for the course STAT303-2-Sec20&Sec21 in Winter 2025. This is not the course text-book. You are required to read the relevant sections of the book as mentioned on the course website.

This book serves as the course notes for [Course Name], and it is an evolving resource developed to support the learning objectives of the course. It builds upon the foundational work of the original iteration, authored and maintained by Professor Arvind Krishna. We are deeply grateful for Professor Krishna's contributions, as his work has provided a robust framework and valuable content upon which this version of the book is based.

As the course progresses during this quarter, the notes will be continually updated and refined to reflect the content taught in real time. The modifications aim to enhance the clarity, depth, and relevance of the material to better align with the current teaching objectives and methodologies.

This book is a living document, and we welcome feedback, suggestions, and contributions from students, instructors, and the broader academic community to help improve its quality and utility.

Thank you for being part of this journey, and we hope this resource serves as a helpful guide throughout the course.

1 Simple Linear Regression

Read section 3.1 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

1.1 Simple Linear Regression

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import statsmodels.api as sm
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
from matplotlib.lines import Line2D
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
```

Develop a simple linear regression model that predicts car price based on engine size. Datasets to be used: *Car_features_train.csv, Car_prices_train.csv*

```
# We are reading training data ONLY at this point.
# Test data is already separated in another file
trainf = pd.read_csv('./Datasets/Car_features_train.csv') # Predictors
trainp = pd.read_csv('./Datasets/Car_prices_train.csv') # Response
train = pd.merge(trainf,trainp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

1.1.1 Training with `statsmodels`

Here, we will use the `statsmodels.formula.api` module of the `statsmodels` library. The use of "API" here doesn't refer to a traditional external web API but rather an interface within the library for users to interact with and perform specific tasks. The `statsmodels.formula.api` module provides a formulaic interface to the `statsmodels` library. A formula is a compact way to specify statistical models using a formula language. This module allows users to define statistical models using formulas similar to those used in R.

So, in summary, the `statsmodels.formula.api` module provides a formulaic interface as part of the `statsmodels` library, allowing users to specify statistical models using a convenient and concise formula syntax.

```
# Let's create the model

# ols stands for Ordinary Least Squares - the name of the algorithm that optimizes Linear Reg

# data input needs the dataframe that has the predictor and the response
# formula input needs to:
    # be a string
    # have the following syntax: "response~predictor"

# Using engineSize to predict price
ols_object = smf.ols(formula = 'price~engineSize', data = train)
```

```
#Using the fit() function of the 'ols' class to fit the model, i.e., train the model
model = ols_object.fit()
```

```
#Printing model summary which contains among other things, the model coefficients
model.summary()
```

Dep. Variable:	price	**R-squared:**	0.390
Model:	OLS	**Adj. R-squared:**	0.390
Method:	Least Squares	**F-statistic:**	3177.
Date:	Tue, 16 Jan 2024	**Prob (F-statistic):**	0.00
Time:	16:46:33	**Log-Likelihood:**	-53949.
No. Observations:	4960	**AIC:**	1.079e+05
Df Residuals:	4958	**BIC:**	1.079e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	-4122.0357	522.260	-7.893	0.000	-5145.896	-3098.176
engineSize	1.299e+04	230.450	56.361	0.000	1.25e+04	1.34e+04

Omnibus:	1271.986	**Durbin-Watson:**	0.517
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	6490.719
Skew:	1.137	**Prob(JB):**	0.00
Kurtosis:	8.122	**Cond. No.**	7.64

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

The model equation is: $\widehat{price}$ = -4122.0357 + 12990 * `engineSize`

- R-squared is 39%. This is the proportion of variance in car price explained by `engineSize`.
- The coef of `engineSize` ($\hat{\beta}_1$) is statistically significant (p-value = 0). There is a linear relationship between X and Y.
- The 95% CI of $\hat{\beta}_1$ is [1.25e+04, 1.34e+04].
- PI is not shown here.

The coefficient of `engineSize` is 1.299e+04. - Unit change in `engineSize` increases the expected price by $ 12,990. - An increase of 3 increases the price by $ (3*1.299e+04) = $ 38,970.

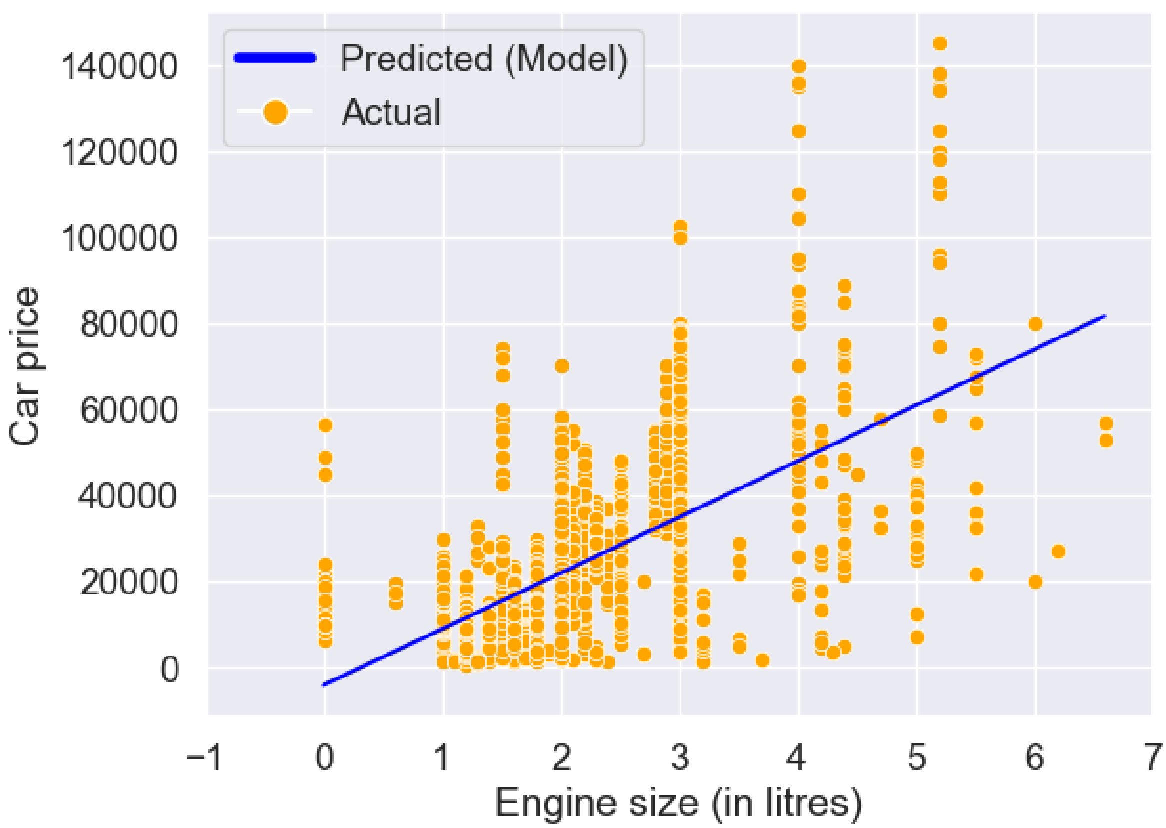
The coefficients can also be returned directly usign the `params` attribute of the `model` object returned by the `fit()` method of the `ols` class:

```
model.params
```

```
Intercept     -4122.035744
engineSize    12988.281021
dtype: float64
```

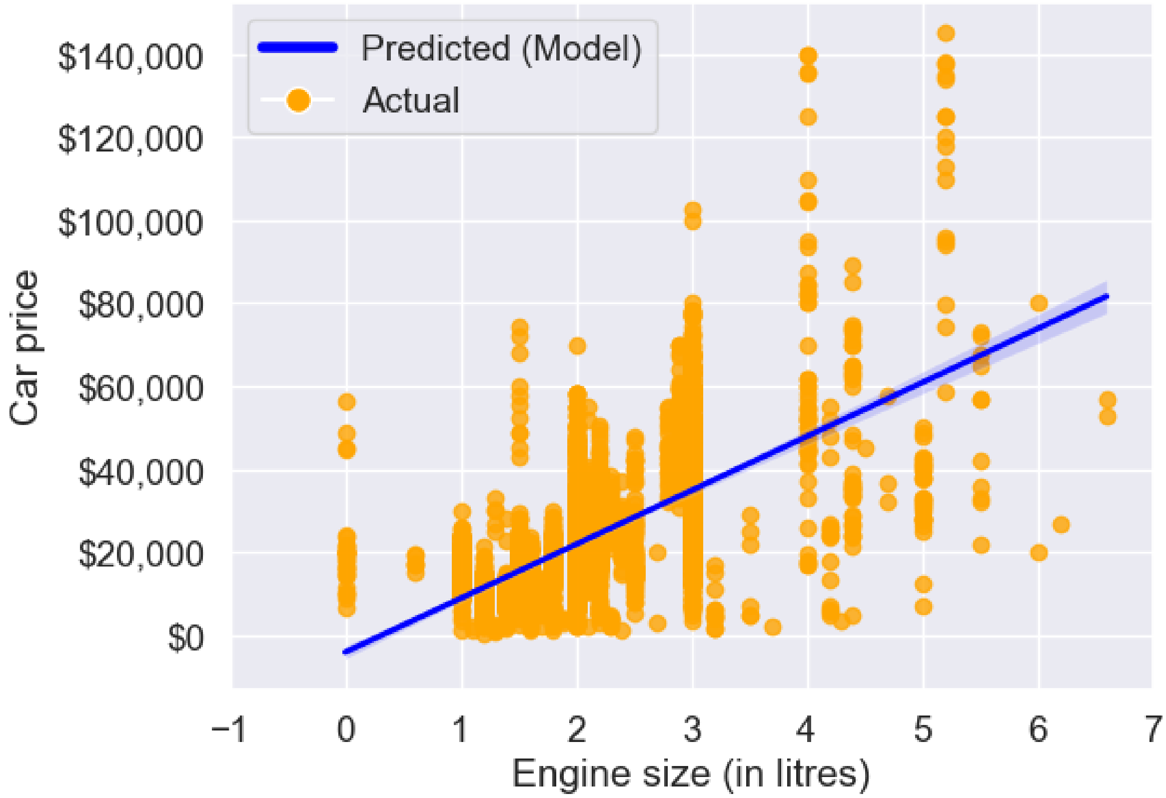
Visualize the regression line

```
sns.set(font_scale=1.25)
ax = sns.scatterplot(x = train.engineSize, y = train.price,color = 'orange')
sns.lineplot(x = train.engineSize, y = model.fittedvalues,color = 'blue')
plt.xlim(-1,7)
plt.xlabel('Engine size (in litres)')
plt.ylabel('Car price')
legend_elements = [Line2D([0], [0], color='blue', lw=4, label='Predicted (Model)'),
                   Line2D([0], [0], marker='o', color='w', label='Actual',
                          markerfacecolor='orange', markersize=10)]
ax.legend(handles=legend_elements, loc='upper left');
```



Note that the above plot can be made directly using the seaborn function `regplot()`. The function `regplot()` fits a simple linear regression model with `y` as the response, and `x` as the predictor, and then plots the model over a scatteplot of the data.

```
ax = sns.regplot(x = 'engineSize', y = 'price', data = train, color = 'orange',line_kws={"col
plt.xlim(-1,7)
plt.xlabel('Engine size (in litres)')
plt.ylabel('Car price')
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.legend(handles=legend_elements, loc='upper left');
#Note that some of the engineSize values are 0. They are incorrect, and should ideally be im
```



The light shaded region around the blue line in the above plot is the confidence interval.

Predict the car price for the cars in the test dataset. Datasets to be used: *Car_features_test.csv, Car_prices_test.csv*

Now that the model has been trained, let us evaluate it on unseen data. Make sure that the columns names of the predictors are the same in train and test datasets.

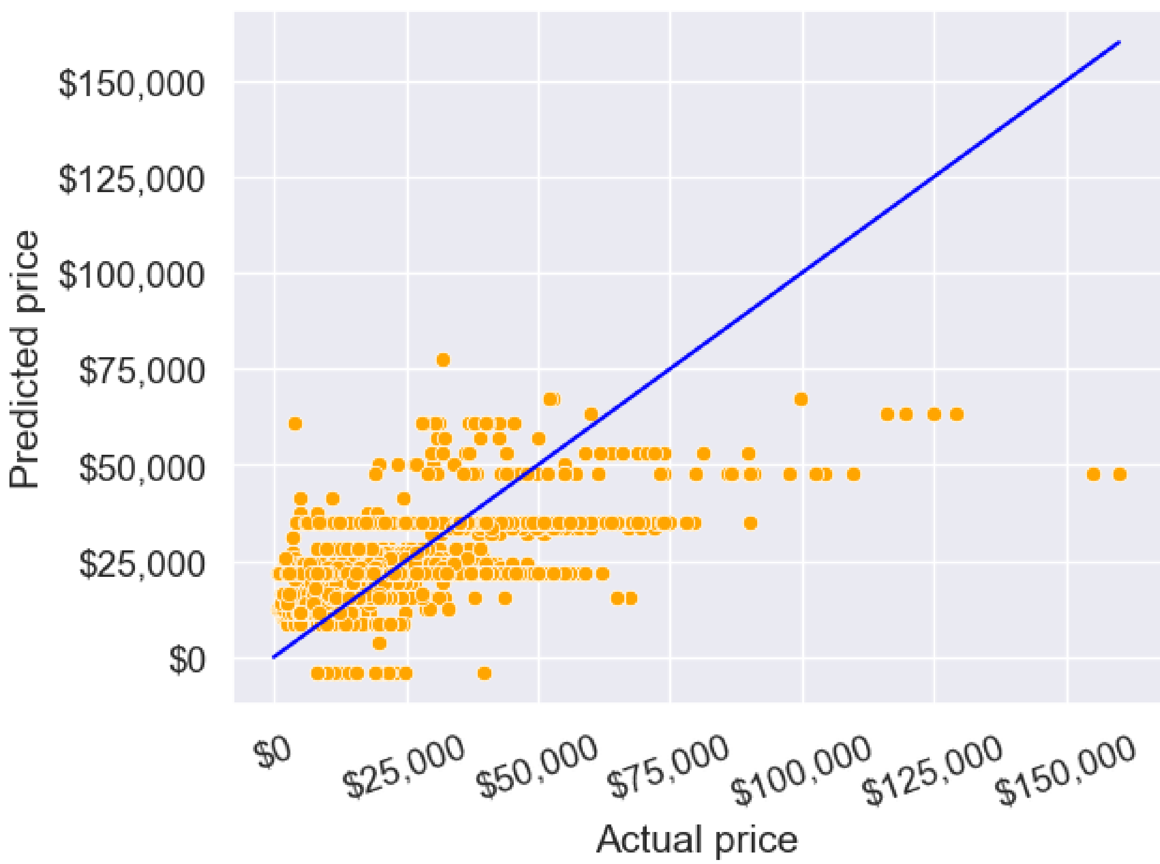
```
# Read the test data
testf = pd.read_csv('./Datasets/Car_features_test.csv') # Predictors
```

```
testp = pd.read_csv('./Datasets/Car_prices_test.csv') # Response
test = pd.merge(testf, testp)
```

```
#Using the predict() function associated with the 'model' object to make predictions of car p
pred_price = model.predict(testf)#Note that the predict() function finds the predictor 'engin
```

Make a visualization that compares the predicted car prices with the actual car prices

```
sns.scatterplot(x = testp.price, y = pred_price, color = 'orange')
#In case of a perfect prediction, all the points must lie on the line x = y.
ax = sns.lineplot(x = [0,testp.price.max()], y = [0,testp.price.max()],color='blue') #Plottin
plt.xlabel('Actual price')
plt.ylabel('Predicted price')
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('${x:,.0f}')
plt.xticks(rotation=20);
```



The prediction doesn't look too good. This is because we are just using one predictor - engine size. We can probably improve the model by adding more predictors when we learn multiple linear regression.

What is the RMSE of the predicted car price on unseen data?

```
np.sqrt(((testp.price - pred_price)**2).mean())
```

```
12995.106451548696
```

The root mean squared error in predicting car price is around $13k.

What is the residual standard error based on the training data?

```
np.sqrt(model.mse_resid)
```

```
12810.109175214138
```

The residual standard error on the training data is close to the RMSE on the test data. This shows that the performance of the model on unknown data is comparable to its performance on known data. This implies that the model is not overfitting, which is good! In case we overfit a model on the training data, its performance on unknown data is likely to be worse than that on the training data.

Find the confidence and prediction intervals of the predicted car price

```
#Using the get_prediction() function associated with the 'model' object to get the intervals
intervals = model.get_prediction(testf)
```

```
#The function requires specifying alpha (probability of Type 1 error) instead of the confiden
intervals.summary_frame(alpha=0.05)
```

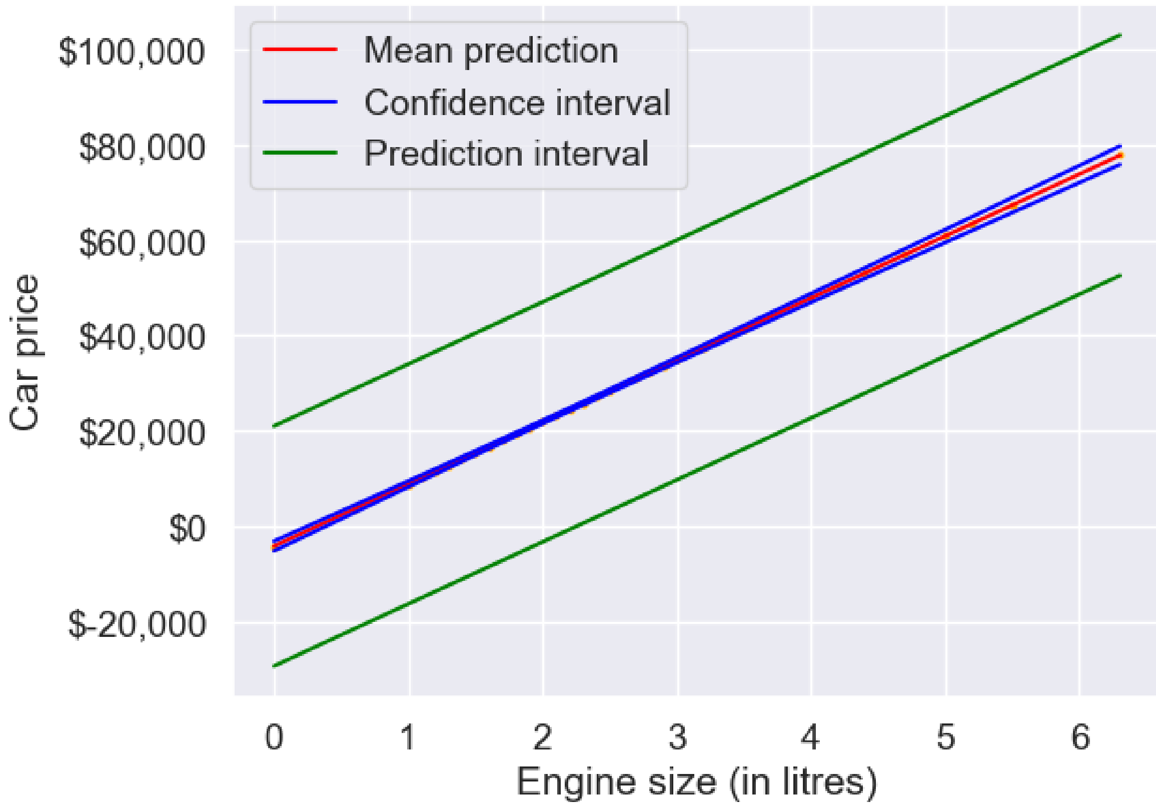
	mean	mean_se	mean_ci_lower	mean_ci_upper	obs_ci_lower	obs_ci_upper
0	34842.807319	271.666459	34310.220826	35375.393812	9723.677232	59961.937406
1	34842.807319	271.666459	34310.220826	35375.393812	9723.677232	59961.937406
2	34842.807319	271.666459	34310.220826	35375.393812	9723.677232	59961.937406
3	8866.245277	316.580850	8245.606701	9486.883853	-16254.905974	33987.396528
4	47831.088340	468.949360	46911.740050	48750.436631	22700.782946	72961.393735
...	...	...	...	...	...	...
2667	47831.088340	468.949360	46911.740050	48750.436631	22700.782946	72961.393735

	mean	mean_se	mean_ci_lower	mean_ci_upper	obs_ci_lower	obs_ci_upper
2668	34842.807319	271.666459	34310.220826	35375.393812	9723.677232	59961.937406
2669	8866.245277	316.580850	8245.606701	9486.883853	-16254.905974	33987.396528
2670	21854.526298	184.135754	21493.538727	22215.513869	-3261.551421	46970.604017
2671	21854.526298	184.135754	21493.538727	22215.513869	-3261.551421	46970.604017

Show the regression line predicting car price based on engine size for test data. Also show the confidence and prediction intervals for the car price.

```
interval_table = intervals.summary_frame(alpha=0.05)
```

```
ax = sns.scatterplot(x = testf.engineSize, y = pred_price,color = 'orange', s = 10)
sns.lineplot(x = testf.engineSize, y = pred_price, color = 'red')
sns.lineplot(x = testf.engineSize, y = interval_table.mean_ci_lower, color = 'blue')
sns.lineplot(x = testf.engineSize, y = interval_table.mean_ci_upper, color = 'blue')
sns.lineplot(x = testf.engineSize, y = interval_table.obs_ci_lower, color = 'green')
sns.lineplot(x = testf.engineSize, y = interval_table.obs_ci_upper, color = 'green')

legend_elements = [Line2D([0], [0], color='red', label='Mean prediction'),
                   Line2D([0], [0], color='blue', label='Confidence interval'),
                  Line2D([0], [0], color='green', label='Prediction interval')]
ax.legend(handles=legend_elements, loc='upper left')
plt.xlabel('Engine size (in litres)')
plt.ylabel('Car price')
ax.yaxis.set_major_formatter('${x:,.0f}');
```



1.1.2 Training with sklearn

```
# Create the model as an object

model = LinearRegression() # No inputs, this will change for other models

# Train the model - separate the predictor(s) and the response for this!
X_train = train[['engineSize']]
y_train = train[['price']]

# Note that both are dfs, NOT series - necessary to avoid errors

model.fit(X_train, y_train)

# Check the slight syntax differences
    # predictors and response separate
    # We need to manually slice the predictor column(s) we want to include
```

```python
    # No need to assign to an output

# Return the parameters
print("Coefficient of engine size = ", model.coef_) # slope
print("Intercept = ", model.intercept_) # intercept

# No .summary() here! - impossible to do much inference; this is a shortcoming of sklearn
```

```
Coefficient of engine size =  [[12988.28102112]]
Intercept =  [-4122.03574424]
```

```python
# Prediction

# Again, separate the predictor(s) and the response of interest
X_test = test[['engineSize']]
y_test = test[['price']].to_numpy() # Easier to handle with calculations as np array

y_pred = model.predict(X_test)

# Evaluate
model_rmse = np.sqrt(np.mean((y_pred - y_test)**2)) # RMSE
model_mae = np.mean(np.abs(y_pred - y_test)) # MAE

print('Test RMSE: ', model_rmse)
```

```
Test RMSE:  12995.106451548696
```

```python
# Easier way to calculate metrics with sklearn tools

# Note that we have imported the functions 'mean_squared_error' and 'mean_absolute_error'
# from the sklearn.metrics module (check top of the code)

model_rmse = np.sqrt(mean_squared_error(y_test,y_pred))
model_mae = mean_absolute_error(y_test,y_pred)
print('Test RMSE: ', model_rmse)
print('Test MAE: ', model_mae)
```

```
Test RMSE:  12995.106451548696
Test MAE:  9411.325912951994
```

```
y_pred_train = model.predict(X_train)
print('Train R-squared:', r2_score(y_train, y_pred_train))
print('Test R-squared:', r2_score(y_test, y_pred))
```

```
Train R-squared: 0.39049842625794573
Test R-squared: 0.3869900378620146
```

Note: Why did we repeat the same task in two different libraries?

- `statsmodels` and `sklearn` have different advantages - we will use both for our purposes
  - `statsmodels` returns a lot of statistical output, which is very helpful for inference (coming up next) but it has a limited variety of models.
  - With `statsmodels`, you may have columns in your DataFrame in addition to predictors and response, while with `sklearn` you need to make separate objects consisting of only the predictors and the response.
  - `sklearn` includes many models (`Lasso` and `Ridge` this quarter, many others next quarter) and helpful tools/functions (like metrics) that statsmodels does not but it does not have any inference tools.

1.1.3 Training with `statsmodels.api`

Earlier we had used the `statsmodels.formula.api` module, where we had to put the regression model as a formula. We can also use the `statsmodels.api` module to develop a regression model. The syntax of training a model with the `OLS()` function in this module is similar to that of `sklearn`'s `LinearRegression()` function. However, the order in which the predictors and response are specified is different. The formula-style syntax of the `statsmodels.formula.api` module is generally preferred. However, depending on the situation, the `OLS()` syntax of `statsmodels.api` may be preferred.

Note that you will manually need to add the predictor *(a column of ones)* corresponding to the intercept to train the model with this method.

```
# Create the model as an object

# Train the model - separate the predictor(s) and the response for this!
X_train = train[['engineSize']]
y_train = train[['price']]

X_train_with_intercept = np.concatenate((np.ones(X_train.shape[0]).reshape(-1,1), X_train), a

model = sm.OLS(y_train, X_train_with_intercept).fit()
```

```python
# Return the parameters
print(model.params)
```

```
const    -4122.035744
x1       12988.281021
dtype: float64
```

The model summary and all other attributes and methods of the `model` object are the same as that with the object created using the `statsmodels.formula.api` module.

```python
model.summary()
```

Dep. Variable:	price	**R-squared:**	0.390
Model:	OLS	**Adj. R-squared:**	0.390
Method:	Least Squares	**F-statistic:**	3177.
Date:	Mon, 08 Jan 2024	**Prob (F-statistic):**	0.00
Time:	11:17:55	**Log-Likelihood:**	-53949.
No. Observations:	4960	**AIC:**	1.079e+05
Df Residuals:	4958	**BIC:**	1.079e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-4122.0357	522.260	-7.893	0.000	-5145.896	-3098.176
x1	1.299e+04	230.450	56.361	0.000	1.25e+04	1.34e+04

Omnibus:	1271.986	**Durbin-Watson:**	0.517
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	6490.719
Skew:	1.137	**Prob(JB):**	0.00
Kurtosis:	8.122	**Cond. No.**	7.64

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

2 Multiple Linear Regression

Read section 3.2 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

2.1 Multiple Linear Regression

```
# importing libraries
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

Develop a multiple linear regression model that predicts car price based on engine size, year, mileage, and mpg. Datasets to be used: *Car_features_train.csv, Car_prices_train.csv*

```
# Reading datasets
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
train = pd.merge(trainf,trainp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

2.1.1 Training the model

```
#Using the ols function to create an ols object. 'ols' stands for 'Ordinary least squares'
ols_object = smf.ols(formula = 'price~year+mileage+mpg+engineSize', data = train)
model = ols_object.fit()
model.summary()
```

Dep. Variable:	price	**R-squared:**	0.660
Model:	OLS	**Adj. R-squared:**	0.660
Method:	Least Squares	**F-statistic:**	2410.
Date:	Mon, 29 Jan 2024	**Prob (F-statistic):**	0.00
Time:	03:10:20	**Log-Likelihood:**	-52497.
No. Observations:	4960	**AIC:**	1.050e+05
Df Residuals:	4955	**BIC:**	1.050e+05
Df Model:	4		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	-3.661e+06	1.49e+05	-24.593	0.000	-3.95e+06	-3.37e+06
year	1817.7366	73.751	24.647	0.000	1673.151	1962.322
mileage	-0.1474	0.009	-16.817	0.000	-0.165	-0.130
mpg	-79.3126	9.338	-8.493	0.000	-97.620	-61.006
engineSize	1.218e+04	189.969	64.107	0.000	1.18e+04	1.26e+04

Omnibus:	2450.973	**Durbin-Watson:**	0.541
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	31060.548
Skew:	2.045	**Prob(JB):**	0.00
Kurtosis:	14.557	**Cond. No.**	3.83e+07

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.83e+07. This might indicate that there are
strong multicollinearity or other numerical problems.

The model equation is: estimated car `price` = -3.661e6 + 1818 * `year` -0.15 * `mileage` - 79.31 * `mpg` + 12180 * `engineSize`

The procedure to fit the model using `sklearn` will be similar to that in simple linear regression.

```
model = LinearRegression()

X_train = train[['year','engineSize','mpg','mileage']] # Slice out the predictors
y_train = train[['price']]

model.fit(X_train,y_train)
```

2.1.2 Hypothesis test for a relationship between the response and a subset of predictors

Let us test the hypothesis if there is relationship between car price and the set of predictors: `mpg` and `year`.

```
hypothesis = '(mpg = 0, year = 0)'

model.f_test(hypothesis) # the F test of these two predictors is stat. sig.
```

```
<class 'statsmodels.stats.contrast.ContrastResults'>
<F test: F=325.9206432972666, p=1.0499509223096256e-133, df_denom=4.96e+03, df_num=2>
```

As the p-value is low, we reject the null hypothesis, i.e., at least one of the predictors among `mpg` and `year` has a statistically significant relationship with car price.

Predict the car price for the cars in the test dataset. Datasets to be used: *Car_features_test.csv, Car_prices_test.csv*

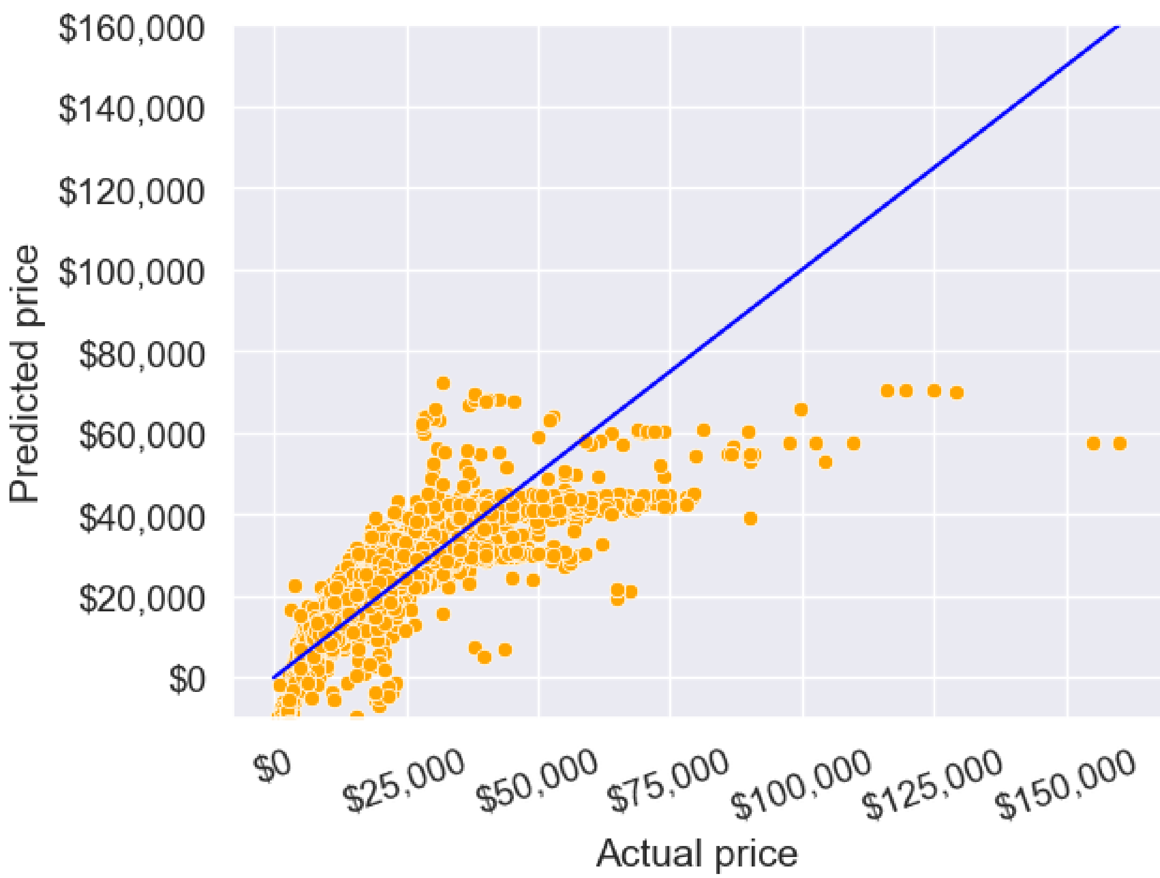
```
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
```

2.1.3 Prediction

```
pred_price = model.predict(testf)
```

Make a visualization that compares the predicted car prices with the actual car prices

```
sns.set(font_scale=1.25)
sns.scatterplot(x = testp.price, y = pred_price, color = 'orange')
#In case of a perfect prediction, all the points must lie on the line x = y.
ax = sns.lineplot(x = [0,testp.price.max()], y = [0,testp.price.max()],color='blue') #Plottin
plt.xlabel('Actual price')
plt.ylabel('Predicted price')
plt.ylim([-10000, 160000])
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('${x:,.0f}')
plt.xticks(rotation=20);
```



The prediction looks better as compared to the one with simple linear regression. This is because we have four predictors to help explain the variation in car price, instead of just one in the case of simple linear regression. Also, all the predictors have a significant relationship with price as evident from their p-values. Thus, all four of them are contributing in explaining the variation. Note the higher values of R^2 as compared to the one in the case of simple linear regression.

What is the RMSE of the predicted car price?

```
np.sqrt(((testp.price - pred_price)**2).mean())
```

```
9956.82497993548
```

What is the residual standard error based on the training data?

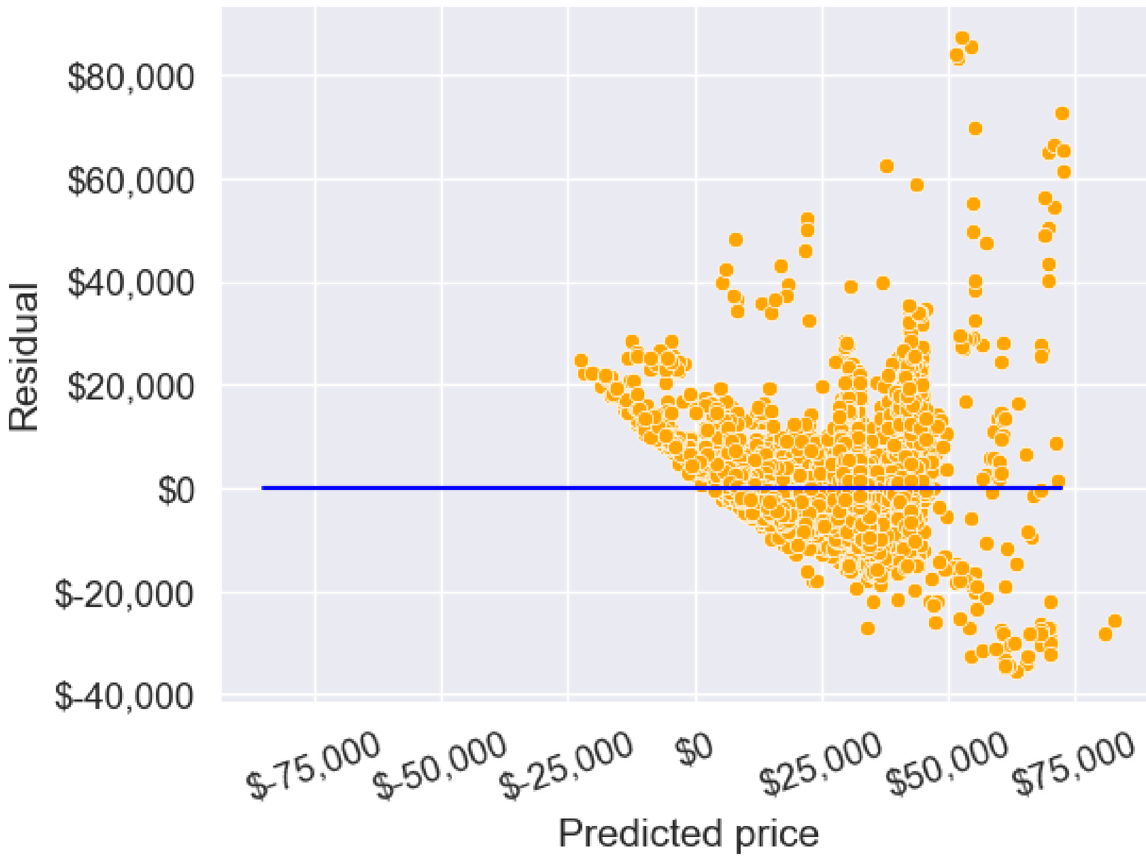
```
np.sqrt(model.mse_resid)
```

```
9563.74782917604
```

```
trainp.describe()
```

	carID	price
count	4960.000000	4960.000000
mean	15832.446169	23469.943750
std	2206.717006	16406.714563
min	12002.000000	450.000000
25%	13929.250000	12000.000000
50%	15840.000000	18999.000000
75%	17765.750000	30335.750000
max	19629.000000	145000.000000

```
sns.scatterplot(x = model.fittedvalues, y=model.resid,color = 'orange')
ax = sns.lineplot(x = [pred_price.min(),pred_price.max()],y = [0,0],color = 'blue')
plt.xlabel('Predicted price')
plt.ylabel('Residual')
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('${x:,.0f}')
plt.xticks(rotation=20);
```



2.1.4 Effect of adding noisy predictors on R^2

Will the explained variation (R-squared) in car price always increase if we add a variable?

Should we keep on adding variables as long as the explained variation (R-squared) is increasing?

```
#Using the ols function to create an ols object. 'ols' stands for 'Ordinary least squares'
np.random.seed(1)
train['rand_col'] = np.random.rand(train.shape[0])
ols_object = smf.ols(formula = 'price~year+mileage+mpg+engineSize+rand_col', data = train)
model = ols_object.fit()
model.summary()
```

Table 2.3: OLS Regression Results

Dep. Variable:	price	R-squared:	0.661
Model:	OLS	Adj. R-squared:	0.660
Method:	Least Squares	F-statistic:	1928.
Date:	Tue, 27 Dec 2022	Prob (F-statistic):	0.00
Time:	01:07:38	Log-Likelihood:	-52497.
No. Observations:	4960	AIC:	1.050e+05
Df Residuals:	4954	BIC:	1.050e+05
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-3.662e+06	1.49e+05	-24.600	0.000	-3.95e+06	-3.37e+06
year	1818.1672	73.753	24.652	0.000	1673.578	1962.756
mileage	-0.1474	0.009	-16.809	0.000	-0.165	-0.130
mpg	-79.2837	9.338	-8.490	0.000	-97.591	-60.976
engineSize	1.218e+04	189.972	64.109	0.000	1.18e+04	1.26e+04
rand_col	451.1226	471.897	0.956	0.339	-474.004	1376.249

Omnibus:	2451.728	Durbin-Watson:	0.541
Prob(Omnibus):	0.000	Jarque-Bera (JB):	31040.331
Skew:	2.046	Prob(JB):	0.00
Kurtosis:	14.552	Cond. No.	3.83e+07

Adding a variable with random values to the model (`rand_col`) increased the explained variation (R^2). This is because the model has one more parameter to tune to reduce the residual squared error (RSS). However, the p-value of `rand_col` suggests that its coefficient is zero. Thus, using the model with `rand_col` may give poorer performance on unknown data, as compared to the model without `rand_col`. This implies that it is not a good idea to blindly add variables in the model to increase R^2.

3 Extending Linear Regression (statsmodels)

Read sections 3.3.1 and 3.3.2 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

3.1 Variable interactions

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

```
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

Until now, we have assumed that the association between a predictor X_j and response Y does not depend on the value of other predictors. For example, the multiple linear regression model that we developed in Chapter 2 assumes that the average increase in price associated with a unit increase in engineSize is always \$12,180, regardless of the value of other predictors. However, this assumption may be incorrect.

3.1.1 Variable interaction between continuous predictors

We can relax this assumption by considering another predictor, called an interaction term. Let us assume that the average increase in `price` associated with a one-unit increase in `engineSize` depends on the model `year` of the car. In other words, there is an interaction between `engineSize` and `year`. This interaction can be included as a predictor, which is the product of `engineSize` and `year`. *Note that there are several possible interactions that we can consider. Here the interaction between `engineSize` and `year` is just an example.*

```
#Considering interaction between engineSize and year
ols_object = smf.ols(formula = 'price~year*engineSize+mileage+mpg', data = train)
model = ols_object.fit()
model.summary()
```

Table 3.2: OLS Regression Results

Dep. Variable:	price	R-squared:	0.682
Model:	OLS	Adj. R-squared:	0.681
Method:	Least Squares	F-statistic:	2121.
Date:	Tue, 24 Jan 2023	Prob (F-statistic):	0.00
Time:	15:28:11	Log-Likelihood:	-52338.
No. Observations:	4960	AIC:	1.047e+05
Df Residuals:	4954	BIC:	1.047e+05
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	5.606e+05	2.74e+05	2.048	0.041	2.4e+04	1.1e+06
year	-275.3833	135.695	-2.029	0.042	-541.405	-9.361
engineSize	-1.796e+06	9.97e+04	-18.019	0.000	-1.99e+06	-1.6e+06
year:engineSize	896.7687	49.431	18.142	0.000	799.861	993.676
mileage	-0.1525	0.008	-17.954	0.000	-0.169	-0.136
mpg	-84.3417	9.048	-9.322	0.000	-102.079	-66.604

Omnibus:	2330.413	Durbin-Watson:	0.524
Prob(Omnibus):	0.000	Jarque-Bera (JB):	29977.437
Skew:	1.908	Prob(JB):	0.00
Kurtosis:	14.423	Cond. No.	7.66e+07

Note that the R-squared has increased as compared to the model in Chapter 2 since we added a predictor.

The model equation is:

$$price = \beta_0+\beta_1*year+\beta_2*engineSize+\beta_3*(year*engineSize)+\beta 4*mileage+\beta_5*mpg, \tag{3.1}$$

or

$$price = \beta_0 + \beta_1 * year + (\beta_2 + \beta_3 * year) * engineSize + \beta 4 * mileage + \beta_5 * mpg, \tag{3.2}$$

or

$$price = \beta_0 + \beta_1 * year + \tilde{\beta} * engineSize + \beta 4 * mileage + \beta_5 * mpg, \tag{3.3}$$

Since $\tilde{\beta}$ is a function of `year`, the association between `engineSize` and `price` is no longer a constant. A change in the value of `year` will change the association between `price` and `engineSize`.

Substituting the values of the coefficients:

`price` = 5.606e5 - 275.3833*`year`* + *(-1.796e6+896.7687`year`)*`engineSize` -0.1525*`mileage`* - *84.3417*`mpg`

Thus, for cars launched in the year 2010, the average increase in price for one liter increase in engine size is -1.796e6 + 896.7687 * 2010 ≈ \$6,500, assuming all the other predictors are constant. However, for cars launched in the year 2020, the average increase in price for one liter increase in engine size is -1.796e6 + 896.7687*2020 ≈ \$15,500 , assuming all the other predictors are constant.

Similarly, the equation can be re-arranged as:

`price` = 5.606e5 +(-275.3833+896.7687*`engineSize`*)`year` -1.796e6*`engineSize`* *-0.1525*`mileage` -84.3417*`mpg`

Thus, for cars with an engine size of 2 litres, the average increase in price for a one year newer model is -275.3833+896.7687 * 2 ≈ \$1500, assuming all the other predictors are constant.

However, for cars with an engine size of 3 litres, the average increase in price for a one year newer model is -275.3833+896.7687 * 3 $\approx$ \$2400, assuming all the other predictors are constant.

```
#Computing the RMSE of the model with the interaction term
pred_price = model.predict(testf)
np.sqrt(((testp.price - pred_price)**2).mean())
```

```
9423.598872501092
```

Note that the RMSE is lower than that of the model in Chapter 2. This is because the interaction term between `engineSize` and `year` is significant and relaxes the assumption of constant association between price and engine size, and between price and year. This added flexibility makes the model better fit the data. Caution: Too much flexibility may lead to overfitting!

Note that interaction terms corresponding to other variable pairs, and higher order interaction terms (such as those containing 3 or 4 variables) may also be significant and improve the model fit & thereby the prediction accuracy of the model.

3.1.2 Including qualitative predictors in the model

Let us develop a model for predicting `price` based on `engineSize` and the qualitative predictor `transmission`.

```
#checking the distribution of values of transmission
train.transmission.value_counts()
```

```
Manual       1948
Automatic    1660
Semi-Auto    1351
Other           1
Name: transmission, dtype: int64
```

Note that the *Other* category of the variable *transmission* contains only a single observation, which is likely to be insufficient to train the model. We'll remove that observation from the training data. Another option may be to combine the observation in the *Other* category with the nearest category, and keep it in the data.

```
train_updated = train[train.transmission!='Other']
```

```
ols_object = smf.ols(formula = 'price ~ engineSize + transmission', data = train_updated)
model = ols_object.fit()
model.summary()
```

Table 3.5: OLS Regression Results

Dep. Variable:	price	R-squared:	0.459
Model:	OLS	Adj. R-squared:	0.458
Method:	Least Squares	F-statistic:	1400.
Date:	Tue, 24 Jan 2023	Prob (F-statistic):	0.00
Time:	15:28:21	Log-Likelihood:	-53644.
No. Observations:	4959	AIC:	1.073e+05
Df Residuals:	4955	BIC:	1.073e+05
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3042.6765	661.190	4.602	0.000	1746.451	4338.902
transmission[T.Manual]	-6770.6165	442.116	-15.314	0.000	-7637.360	-5903.873
transmission[T.Semi-Auto]	4994.3112	442.989	11.274	0.000	4125.857	5862.765
engineSize	1.023e+04	247.485	41.323	0.000	9741.581	1.07e+04

Omnibus:	1575.518	Durbin-Watson:	0.579
Prob(Omnibus):	0.000	Jarque-Bera (JB):	11006.609
Skew:	1.334	Prob(JB):	0.00
Kurtosis:	9.793	Cond. No.	11.4

Note that there is no coefficient for the *Automatic* level of the variable `Transmission`. If a car doesn't have *Manual* or *Semi-Automatic* transmission, then it has an *Automatic* transmission. Thus, the coefficient of *Automatic* will be redundant, and the dummy variable corresponding to *Automatic* transmission is dropped from the model.

The level of the categorical variable that is dropped from the model is called the baseline level. Here *Automatic* transmission is the baseline level. The coefficients of other levels of `transmission` should be interpreted with respect to the baseline level.

Q: Interpret the intercept term

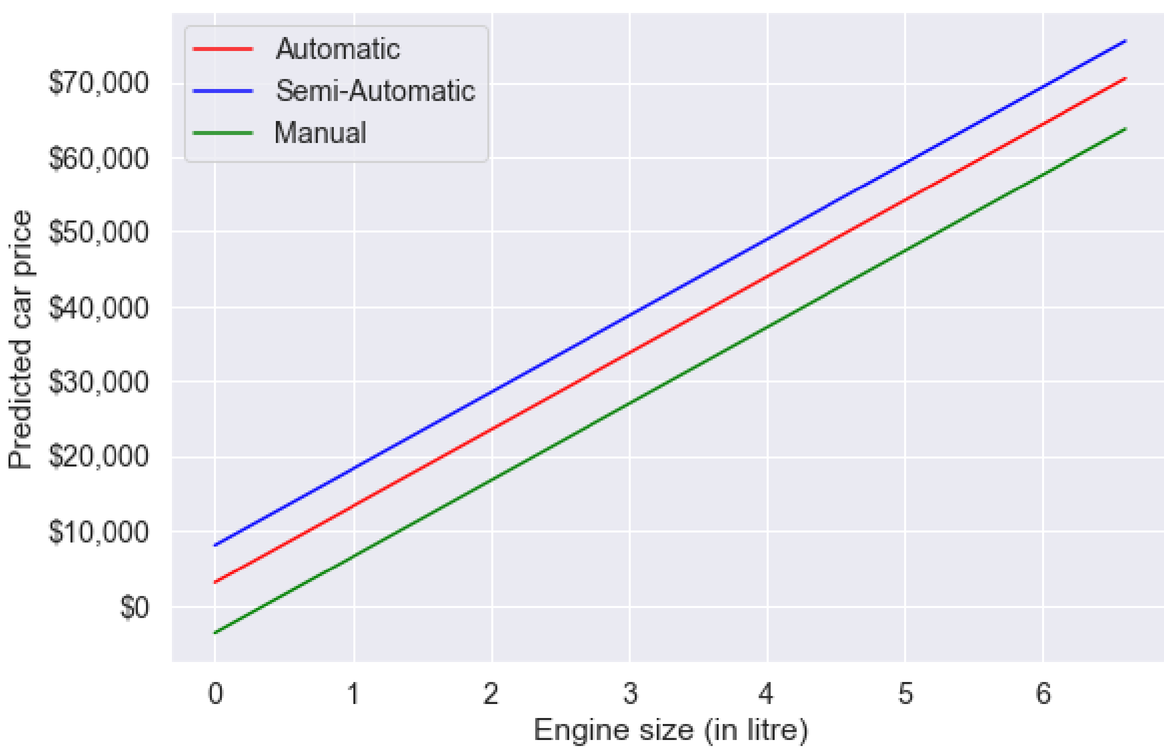
Ans: For the hypothetical scenario of a car with zero engine size and *Automatic* transmission, the estimated mean car price is $\approx$ \$3042.

Q: Interpret the coefficient of `transmission[T.Manual]`

Ans: The estimated mean price of a car with manual transmission is $\approx$ \$6770 less than that of a car with *Automatic* transmission.

Let us visualize the developed model.

```
#Visualizing the developed model
plt.rcParams["figure.figsize"] = (9,6)
sns.set(font_scale = 1.3)
x = np.linspace(train_updated.engineSize.min(),train_updated.engineSize.max(),100)
ax = sns.lineplot(x = x, y = model.params['engineSize']*x+model.params['Intercept'], color =
sns.lineplot(x = x, y = model.params['engineSize']*x+model.params['Intercept']+model.params[
sns.lineplot(x = x, y = model.params['engineSize']*x+model.params['Intercept']+model.params[
plt.legend(labels=["Automatic","Semi-Automatic", "Manual"])
plt.xlabel('Engine size (in litre)')
plt.ylabel('Predicted car price')
ax.yaxis.set_major_formatter('${x:,.0f}')
```



Based on the developed model, for a given engine size, the car with a semi-automatic transmission is estimated to be the most expensive on average, while the car with a manual transmission is estimated to be the least expensive on average.

Changing the baseline level: By default, the baseline level is chosen as the one that comes first if the levels are arranged in alphabetical order. However, you can change the baseline level by specifying one explicitly.

Internally, statsmodels uses the patsy package to convert formulas and data to the matrices that are used in model fitting. You may refer to this section in the patsy documentation to specify a particular level of the categorical variable as the baseline.

For example, suppose we wish to change the baseline level to *Manual* transmission. We can specify this in the formula as follows:

```
ols_object = smf.ols(formula = 'price~engineSize+C(transmission, Treatment("Manual"))', data
model = ols_object.fit()
model.summary()
```

Table 3.8: OLS Regression Results

Dep. Variable:	price	R-squared:	0.459
Model:	OLS	Adj. R-squared:	0.458
Method:	Least Squares	F-statistic:	1400.
Date:	Tue, 24 Jan 2023	Prob (F-statistic):	0.00
Time:	15:28:39	Log-Likelihood:	-53644.
No. Observations:	4959	AIC:	1.073e+05
Df Residuals:	4955	BIC:	1.073e+05
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.9
Intercept	-3727.9400	492.917	-7.563	0.000	-4694.275	-27
C(transmission, Treatment("Manual"))[T.Automatic]	6770.6165	442.116	15.314	0.000	5903.873	763
C(transmission, Treatment("Manual"))[T.Semi-Auto]	1.176e+04	473.110	24.867	0.000	1.08e+04	1.2
engineSize	1.023e+04	247.485	41.323	0.000	9741.581	1.0

Omnibus:	1575.518	Durbin-Watson:	0.579
Prob(Omnibus):	0.000	Jarque-Bera (JB):	11006.609
Skew:	1.334	Prob(JB):	0.00
Kurtosis:	9.793	Cond. No.	8.62

3.1.3 Including qualitative predictors and their interaction with continuous predictors in the model

Note that the qualitative predictor leads to fitting 3 parallel lines to the data, as there are 3 categories.

However, note that we have made the constant association assumption. The fact that the lines are parallel means that the average increase in car price for one litre increase in engine size does not depend on the type of transmission. This represents a potentially serious limitation of the model, since in fact a change in engine size may have a very different association on the price of an automatic car versus a semi-automatic or manual car.

This limitation can be addressed by adding an interaction variable, which is the product of `engineSize` and the dummy variables for semi-automatic and manual transmissions.

```
#Using the ols function to create an ols object. 'ols' stands for 'Ordinary least squares'
ols_object = smf.ols(formula = 'price~engineSize*transmission', data = train_updated)
model = ols_object.fit()
model.summary()
```

Table 3.11: OLS Regression Results

Dep. Variable:	price	R-squared:	0.479
Model:	OLS	Adj. R-squared:	0.478
Method:	Least Squares	F-statistic:	909.9
Date:	Sun, 22 Jan 2023	Prob (F-statistic):	0.00
Time:	22:55:55	Log-Likelihood:	-53550.
No. Observations:	4959	AIC:	1.071e+05
Df Residuals:	4953	BIC:	1.072e+05
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3754.7238	895.221	4.194	0.000	1999.695	5509.753
transmission[T.Manual]	1768.5856	1294.071	1.367	0.172	-768.366	4305.538
transmission[T.Semi-Auto]	-5282.7164	1416.472	-3.729	0.000	-8059.628	-2505.805
engineSize	9928.6082	354.511	28.006	0.000	9233.610	1.06e+04
engineSize:transmission[T.Manual]	-5285.9059	646.175	-8.180	0.000	-6552.695	-4019.117
engineSize:transmission[T.Semi-Auto]	4162.2428	552.597	7.532	0.000	3078.908	5245.578

Omnibus:	1379.846	Durbin-Watson:	0.622
Prob(Omnibus):	0.000	Jarque-Bera (JB):	9799.471
Skew:	1.139	Prob(JB):	0.00
Kurtosis:	9.499	Cond. No.	30.8

The model equation for the model with interactions is:

Automatic transmission: `price` = 3754.7238 + 9928.6082 * `engineSize`,

Semi-Automatic transmission: `price` = 3754.7238 + 9928.6082 * `engineSize` + (-5282.7164+4162.2428*`engineSize`),

Manual transmission: `price` = 3754.7238 + 9928.6082 * `engineSize` +(1768.5856-5285.9059 * `engineSize`),

or

Automatic transmission: `price` = 3754.7238 + 9928.6082 * `engineSize`,

Semi-Automatic transmission: `price` = -1527 + 7046 * `engineSize`,

Manual transmission: `price` = 5523 + 4642 * `engineSize`

Q: Interpret the coefficient of manual tranmission, i.e., the coefficient of `transmission[T.Manual]`.

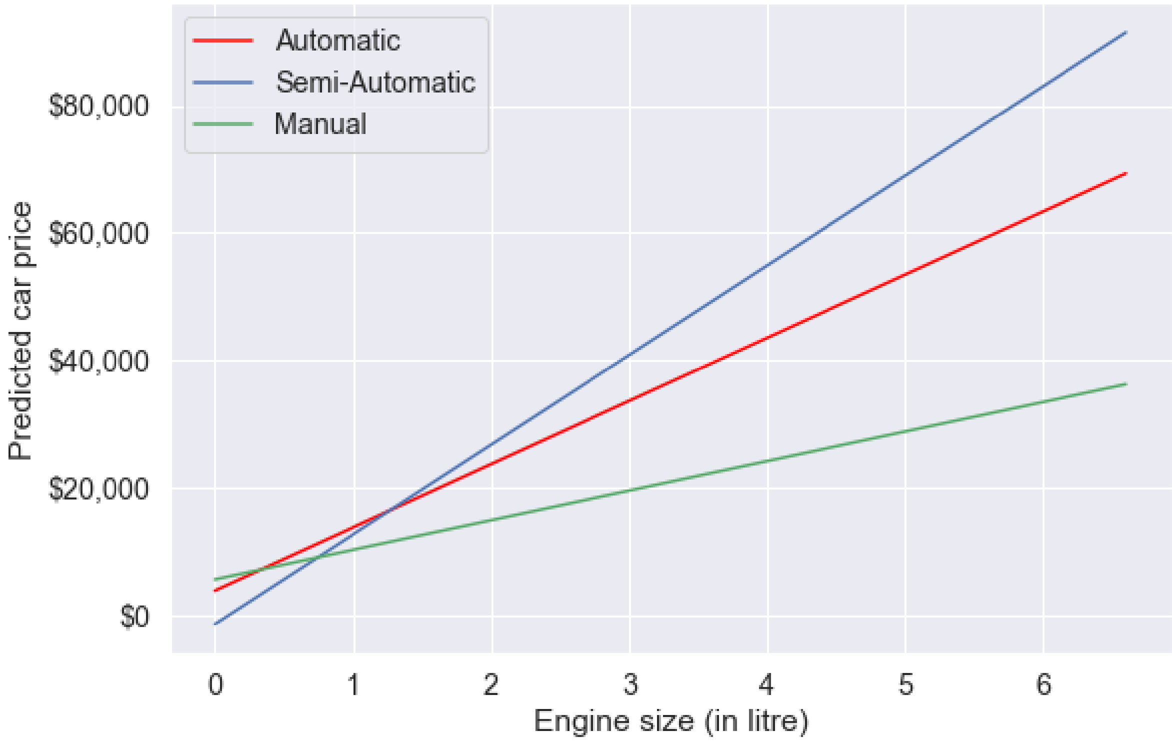
A: For a hypothetical scenario of zero engine size, the estimated mean `price` of a car with *Manual* `transmission` is ≈ \$1768 more than the estimated mean `price` of a car with *Automatic* `transmission`.

Q: Interpret the coefficient of the interaction between engine size and manual transmission, i.e., the coefficient of `engineSize:transmission[T.Manual]`.

A: For a unit (or a litre) increase in `engineSize` , the increase in estimated mean `price` of a car with *Manual* transmission is ≈ \$5285 less than the increase in estimated mean `price` of a car with *Automatic* transmission.

```
#Visualizing the developed model with interaction terms
plt.rcParams["figure.figsize"] = (9,6)
sns.set(font_scale = 1.3)
x = np.linspace(train_updated.engineSize.min(),train_updated.engineSize.max(),100)
ax = sns.lineplot(x = x, y = model.params['engineSize']*x+model.params['Intercept'], label='A
plt.plot(x, (model.params['engineSize']+model.params['engineSize:transmission[T.Semi-Auto]']
plt.plot(x, (model.params['engineSize']+model.params['engineSize:transmission[T.Manual]'])*x+
plt.legend(loc='upper left')
plt.xlabel('Engine size (in litre)')
```

```
plt.ylabel('Predicted car price')
ax.yaxis.set_major_formatter('${x:,.0f}')
```



Note the interaction term adds flexibility to the model.

The slope of the regression line for semi-automatic cars is the largest. This suggests that increase in engine size is associated with a higher increase in car price for semi-automatic cars, as compared to other cars.

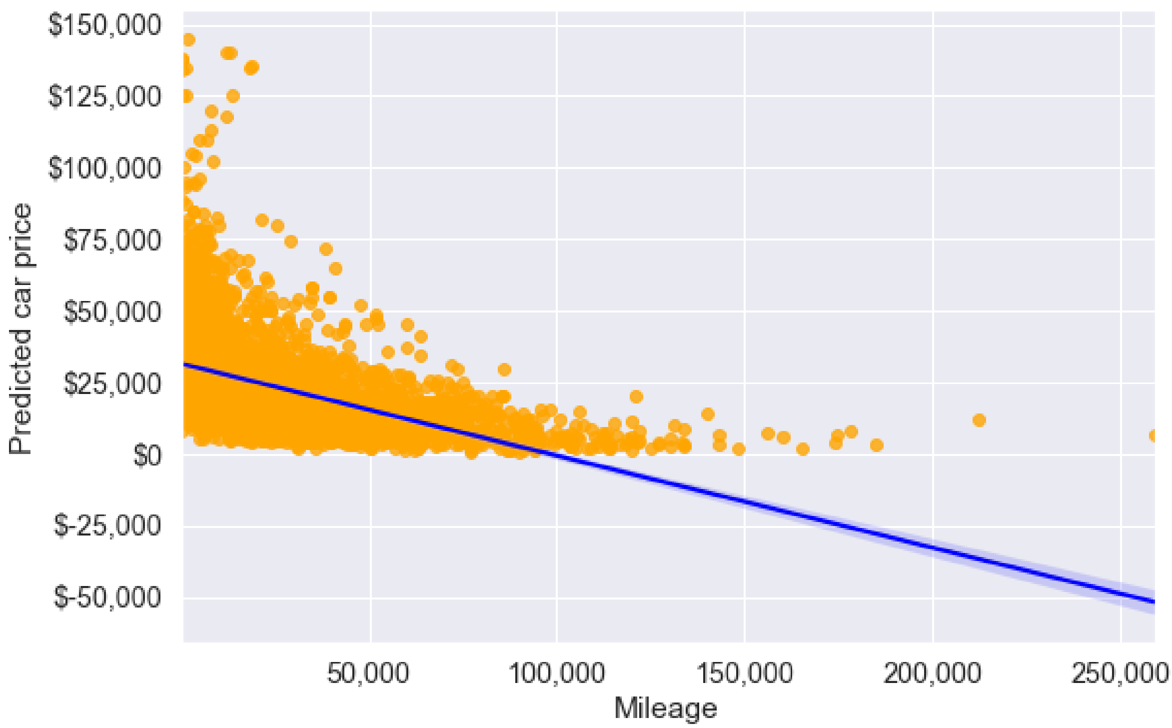
3.2 Variable transformations

So far we have considered only a linear relationship between the predictors and the response. However, the relationship may be non-linear.

Consider the regression plot of `price` on `mileage`.

```
ax = sns.regplot(x = train_updated.mileage, y =train_updated.price,color = 'orange', line_kws
plt.xlabel('Mileage')
plt.ylabel('Predicted car price')
```

```
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('{x:,.0f}')
```




```
#R-squared of the model with just mileage
model = smf.ols('price~mileage', data = train_updated).fit()
model.rsquared
```

```
0.22928048993376182
```

From the first scatterplot, we see that the relationship between `price` and `mileage` doesn't seem to be linear, as the points do not lie on a straight line. Also, we see the regression line (or the curve), which is the best fit line doesn't seem to fit the points well. However, `price` on average seems to decrease with `mileage`, albeit in a non-linear manner.

3.2.1 Quadratic transformation

So, we guess that if we model price as a quadratic function of `mileage`, the model may better fit the points (or the curve may better fit the points). Let us transform the predictor `mileage` to include $mileage^2$ (i.e., perform a quadratic transformation on the predictor).

```
#Including mileage squared as a predictor and developing the model
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)', data = train_updated)
model = ols_object.fit()
model.summary()
```

Table 3.14: OLS Regression Results

Dep. Variable:	price	R-squared:	0.271
Model:	OLS	Adj. R-squared:	0.271
Method:	Least Squares	F-statistic:	920.6
Date:	Sun, 22 Jan 2023	Prob (F-statistic):	0.00
Time:	23:26:05	Log-Likelihood:	-54382.
No. Observations:	4959	AIC:	1.088e+05
Df Residuals:	4956	BIC:	1.088e+05
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.44e+04	332.710	103.382	0.000	3.37e+04	3.5e+04
mileage	-0.5662	0.017	-33.940	0.000	-0.599	-0.534
I(mileage ** 2)	2.629e-06	1.56e-07	16.813	0.000	2.32e-06	2.94e-06

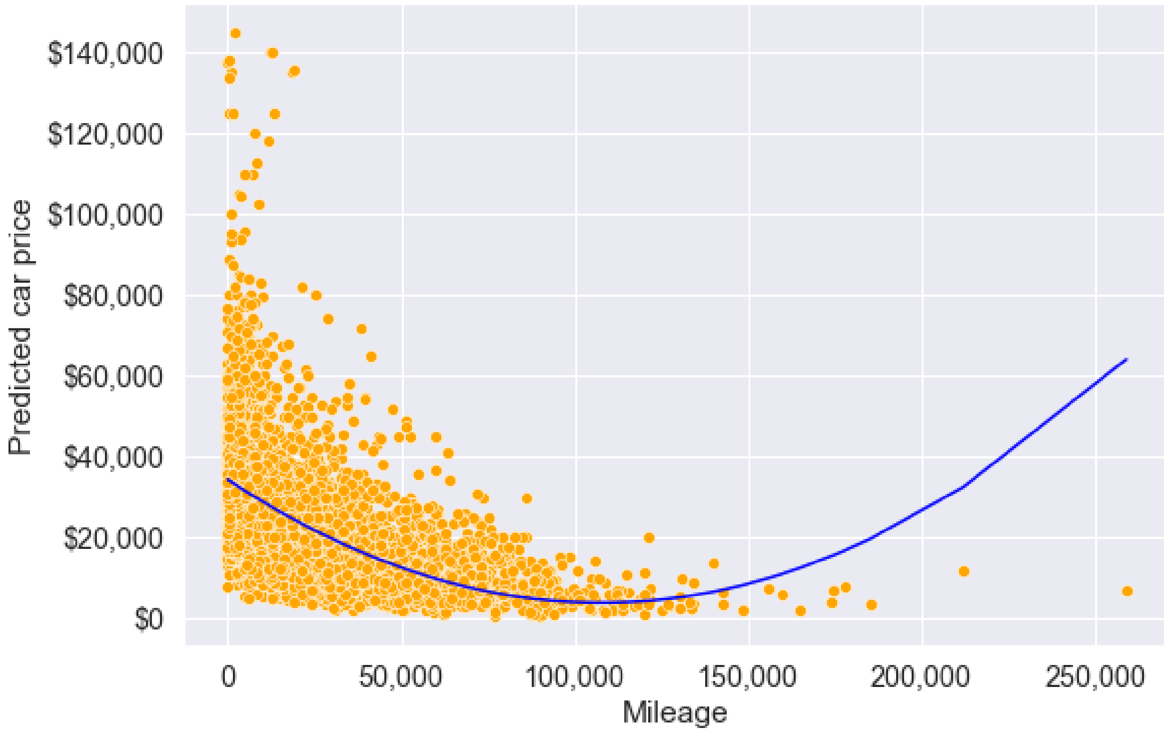
Omnibus:	2362.973	Durbin-Watson:	0.325
Prob(Omnibus):	0.000	Jarque-Bera (JB):	22427.952
Skew:	2.052	Prob(JB):	0.00
Kurtosis:	12.576	Cond. No.	4.81e+09

Note that in the formula specified within the `ols()` function, the `I()` operator isolates or insulates the contents within I(…) from the regular formula operators. Without the `I()` operator, `mileage**2` will be treated as the interaction of `mileage` with itself, which is `mileage`. Thus, to add the square of `mileage` as a separate predictor, we need to use the `I()` operator.

Let us visualize the model fit with the quadratic transformation of the predictor - `mileage`.

```
#Visualizing the regression line with the model consisting of the quadratic transformation o
pred_price = model.predict(train_updated)
ax = sns.scatterplot(x = 'mileage', y = 'price', data = train_updated, color = 'orange')
sns.lineplot(x = train_updated.mileage, y = pred_price, color = 'blue')
plt.xlabel('Mileage')
```

```
plt.ylabel('Predicted car price')
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('{x:,.0f}')
```



The above model seems to better fit the data (as compared to the model without transformation) at least upto mileage around 125,000. The R^2 of the model with the quadratic transformation of `mileage` is also higher than that of the model without transformation indicating a better fit.

3.2.2 Cubic transformation

Let us see if a cubic transformation of `mileage` can further improve the model fit.

```
#Including mileage squared and mileage cube as predictors and developing the model
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)+I(mileage**3)', data = train_upda
model = ols_object.fit()
model.summary()
```

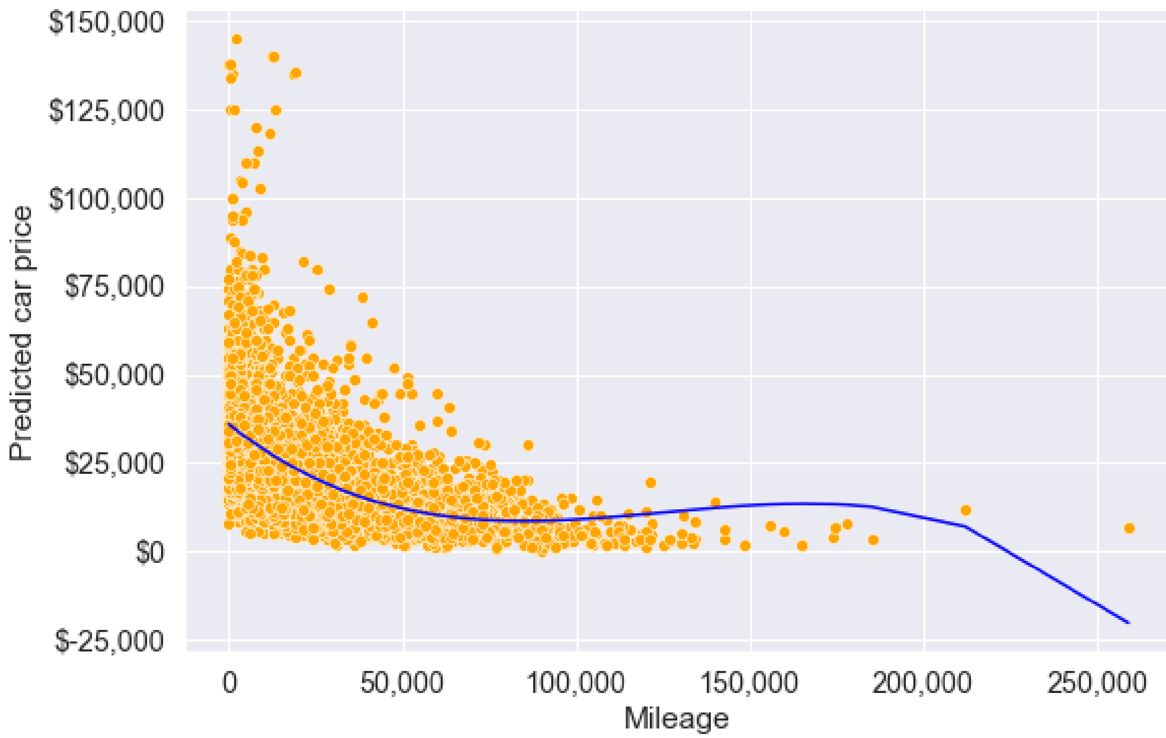
Table 3.17: OLS Regression Results

Dep. Variable:	price	R-squared:	0.283
Model:	OLS	Adj. R-squared:	0.283
Method:	Least Squares	F-statistic:	652.3
Date:	Sun, 22 Jan 2023	Prob (F-statistic):	0.00
Time:	23:33:27	Log-Likelihood:	-54340.
No. Observations:	4959	AIC:	1.087e+05
Df Residuals:	4955	BIC:	1.087e+05
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.598e+04	371.926	96.727	0.000	3.52e+04	3.67e+04
mileage	-0.7742	0.028	-27.634	0.000	-0.829	-0.719
I(mileage ** 2)	6.875e-06	4.87e-07	14.119	0.000	5.92e-06	7.83e-06
I(mileage ** 3)	-1.823e-11	1.98e-12	-9.199	0.000	-2.21e-11	-1.43e-11

Omnibus:	2380.788	Durbin-Watson:	0.321
Prob(Omnibus):	0.000	Jarque-Bera (JB):	23039.307
Skew:	2.065	Prob(JB):	0.00
Kurtosis:	12.719	Cond. No.	7.73e+14

```
#Visualizing the model with the cubic transformation of mileage
pred_price = model.predict(train_updated)
ax = sns.scatterplot(x = 'mileage', y = 'price', data = train_updated, color = 'orange')
sns.lineplot(x = train_updated.mileage, y = pred_price, color = 'blue')
plt.xlabel('Mileage')
plt.ylabel('Predicted car price')
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('{x:,.0f}')
```



Note that the model fit with the cubic transformation of `mileage` seems slightly better as compared to the models with the quadratic transformation, and no transformation of mileage, for mileage up to 180k. However, the model should not be used to predict car prices of cars with a mileage higher than 180k.

Let's update the model created earlier (in the beginning of this chapter) to include the transformed predictor.

```
#Model with an interaction term and a variable transformation term
ols_object = smf.ols(formula = 'price~year*engineSize+mileage+mpg+I(mileage**2)', data = tra
model = ols_object.fit()
model.summary()
```

Table 3.20: OLS Regression Results

Dep. Variable:	price	R-squared:	0.702
Model:	OLS	Adj. R-squared:	0.702
Method:	Least Squares	F-statistic:	1947.
Date:	Sun, 22 Jan 2023	Prob (F-statistic):	0.00
Time:	23:42:13	Log-Likelihood:	-52162.
No. Observations:	4959	AIC:	1.043e+05

Df Residuals:	4952	BIC:	1.044e+05
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.53e+06	2.7e+05	5.671	0.000	1e+06	2.06e+06
year	-755.7419	133.791	-5.649	0.000	-1018.031	-493.453
engineSize	-2.022e+06	9.72e+04	-20.803	0.000	-2.21e+06	-1.83e+06
year:engineSize	1008.6993	48.196	20.929	0.000	914.215	1103.184
mileage	-0.3548	0.014	-25.973	0.000	-0.382	-0.328
mpg	-54.7450	8.896	-6.154	0.000	-72.185	-37.305
I(mileage ** 2)	1.926e-06	1.04e-07	18.536	0.000	1.72e-06	2.13e-06

Omnibus:	2355.448	Durbin-Watson:	0.562
Prob(Omnibus):	0.000	Jarque-Bera (JB):	38317.404
Skew:	1.857	Prob(JB):	0.00
Kurtosis:	16.101	Cond. No.	6.40e+12

Note that the R-squared has increased as compared to the model with just the interaction term.

```
#Computing RMSE on test data
pred_price = model.predict(testf)
np.sqrt(((testp.price - pred_price)**2).mean())
```

```
9074.494088619422
```

Note that the prediction accuracy of the model has further increased, as the RMSE has reduced. The transformed predictor is statistically significant and provides additional flexibility to better capture the trend in the data, leading to an increase in prediction accuracy.

3.3 `PolynomialFeatures()`

The function `PolynomialFeatures()` from the `sklearn` library can be used to generate a predictor matrix that includes all interactions and transformations upto a degree `d`.

```
X_train = train[['mileage', 'engineSize', 'year', 'mpg']]
y_train = train[['price']]
X_test = test[['mileage', 'engineSize', 'year', 'mpg']]
y_test = test[['price']]
```

3.3.1 Generating polynomial features

Let us generate polynomial features upto degree 2. This will include all the two-factor interactions, and all squared terms of degree 2.

```
poly = PolynomialFeatures(2, include_bias = False) # Create the object - degree is 2

# Generate the polynomial features
X_train_poly = poly.fit_transform(X_train)
```

Note that the `LinearRegression()` function adds the intercept by default *(check the* *`fit_intercept`* *argument)*. Thus, we have put `include_bias = False` while generating the polynomial features, as we don't need the intercept. The term `bias` here refers to the intercept *(you will learn about* *`bias`* *in detail in STAT303-3)*. Another option is to include the intercept while generating the polynomial features, and put `fit_intercept = False` in the `LinearRegression()` function.

Below are the polynomial features generated by the `PolynomialFeatures()` functions.

```
poly.get_feature_names_out()
```

```
array(['mileage', 'engineSize', 'year', 'mpg', 'mileage^2',
       'mileage engineSize', 'mileage year', 'mileage mpg',
       'engineSize^2', 'engineSize year', 'engineSize mpg', 'year^2',
       'year mpg', 'mpg^2'], dtype=object)
```

3.3.2 Fitting the model

```
model = LinearRegression()
model.fit(X_train_poly, y_train)
```

```
LinearRegression()
```

3.3.3 Testing the model

```
X_test_poly = poly.fit_transform(X_test)
```

```
#RMSE
np.sqrt(mean_squared_error(y_test, model.predict(X_test_poly)))
```

```
8896.175508213777
```

Note that the polynomial features have helped reduced the RMSE further.

4 Extending Linear Regression (PolynomialFeatures in Sklearn)

4.0.1 Simulate Data

```
import numpy as np
import pandas as pd

# Set a random seed for reproducibility
np.random.seed(42)

# Number of samples
N = 5000

# Generate features from uniform distributions
x1 = np.random.uniform(-5, 5, N)
x2 = np.random.uniform(-5, 5, N)

# Define the nonlinear relationship and add noise
y = 1.5 * (x1 ** 2) + 0.5 * (x2 ** 3) + np.random.normal(loc=3, scale=3, size=N)

# Create a pandas DataFrame
df = pd.DataFrame({'x1': x1, 'x2': x2, 'y': y})

# Save to CSV (optional)
df.to_csv('nonlinear_dataset.csv', index=False)

df.head(10)  # Display the first 10 rows
```

	x1	x2	y
0	-1.254599	-1.063645	0.295770
1	4.507143	-0.265643	30.086577
2	2.319939	3.545474	34.523622

	x1	x2	y
3	0.986585	-1.599956	-1.109427
4	-3.439814	3.696497	49.341010
5	-3.440055	-4.118656	-14.395442
6	-4.419164	2.767984	43.154083
7	3.661761	3.475476	43.267654
8	1.011150	-3.181823	-9.254211
9	2.080726	-0.696535	11.674644

```
# Create X and y arrays
X = df[['x1', 'x2']].values
y = df['y'].values
```

4.0.2 Train-Test Split

```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Training set shape:", X_train.shape)
print("Testing set shape:", X_test.shape)
```

```
Training set shape: (4000, 2)
Testing set shape: (1000, 2)
```

4.0.3 Baseline Model (original Features)

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Create a linear regression model
baseline_model = LinearRegression()

# Train the model on the original features
baseline_model.fit(X_train, y_train)
```

```python
# Make predictions on the test set
y_pred_baseline = baseline_model.predict(X_test)

# Make predictions on the training set
y_train_pred_baseline = baseline_model.predict(X_train)

# Evaluate the baseline model
mse_baseline = mean_squared_error(y_test, y_pred_baseline)
r2_baseline = r2_score(y_test, y_pred_baseline)

# Evaluate the baseline model on the training set
mse_train_baseline = mean_squared_error(y_train, y_train_pred_baseline)
r2_train_baseline = r2_score(y_train, y_train_pred_baseline)

print("\nBaseline Model Performance:")
print("Training Set:")
print("MSE:", mse_train_baseline)
print("R2:", r2_train_baseline)
print("\nTesting Set:")
print("MSE:", mse_baseline)
print("R2:", r2_baseline)
```

```
Baseline Model Performance:
Training Set:
MSE: 218.84992612063238
R2: 0.6720857612554578

Testing Set:
MSE: 218.05222660659857
R2: 0.6745129537467693
```

4.0.4 Transform Features with PolynomialFeatures (`degree = 2`)

```python
from sklearn.preprocessing import PolynomialFeatures

# Create PolynomialFeatures object with degree=2 (includes interaction terms)
poly_2 = PolynomialFeatures(degree=2, include_bias=False)
```

```python
# Transform the training and testing features
X_train_poly_2 = poly_2.fit_transform(X_train)
X_test_poly_2 = poly_2.transform(X_test)

# Display the transformed feature names
print("\nTransformed Feature Names:")
print(poly_2.get_feature_names_out())
```

```
Transformed Feature Names:
['x0' 'x1' 'x0^2' 'x0 x1' 'x1^2']
```

4.0.5 Linear Model with transformed Features (`degree = 2`)

```python
# Create a linear regression model for the polynomial features
poly_2_model = LinearRegression()

# Train the model on the transformed features
poly_2_model.fit(X_train_poly_2, y_train)

# Make predictions on the test set
y_pred_poly_2 = poly_2_model.predict(X_test_poly_2)

# Make predictions on the training set
y_train_pred_poly_2 = poly_2_model.predict(X_train_poly_2)

# Evaluate the polynomial model
mse_poly_2 = mean_squared_error(y_test, y_pred_poly_2)
r2_poly_2 = r2_score(y_test, y_pred_poly_2)

# Evaluate the polynomial model on the training set
mse_train_poly_2 = mean_squared_error(y_train, y_train_pred_poly_2)
r2_train_poly_2 = r2_score(y_train, y_train_pred_poly_2)

print("\nPolynomial Model Performance:")
print("Training Set:")
print("MSE:", mse_train_poly_2)
print("R2:", r2_train_poly_2)
print("\nTesting Set:")
```

```
print("MSE:", mse_poly_2)
print("R2:", r2_poly_2)
```

```
Polynomial Model Performance:
Training Set:
MSE: 95.31378244224537
R2: 0.8571863972474734

Testing Set:
MSE: 90.00139440016171
R2: 0.8656547173222298
```

4.0.6 Transform features with PolynomialFeatures (`degree = 3`)

```
# Create PolynomialFeatures object with degree=2 (includes interaction terms)
poly_3 = PolynomialFeatures(degree=3, include_bias=False)

# Transform the training and testing features
X_train_poly_3 = poly_3.fit_transform(X_train)
X_test_poly_3 = poly_3.transform(X_test)

# Display the transformed feature names
print("\nTransformed Feature Names:")
print(poly_3.get_feature_names_out())
```

```
Transformed Feature Names:
['x0' 'x1' 'x0^2' 'x0 x1' 'x1^2' 'x0^3' 'x0^2 x1' 'x0 x1^2' 'x1^3']
```

4.0.7 Linear Model with transformed Features (`degree = 3`)

```
# Create a linear regression model for the polynomial features
poly_3_model = LinearRegression()

# Train the model on the transformed features
poly_3_model.fit(X_train_poly_3, y_train)
```

```python
# Make predictions on the test set
y_pred_poly_3 = poly_3_model.predict(X_test_poly_3)

# Make predictions on the training set
y_pred_train_poly_3 = poly_3_model.predict(X_train_poly_3)

# Evaluate the polynomial model
mse_poly_3 = mean_squared_error(y_test, y_pred_poly_3)
r2_poly_3 = r2_score(y_test, y_pred_poly_3)

# Evaluate the polynomial model on the training set
mse_poly_3_train = mean_squared_error(y_train, y_pred_train_poly_3)
r2_poly_3_train = r2_score(y_train, y_pred_train_poly_3)

print("\nPolynomial Model Performance:")
print("Training Set:")
print("MSE:", mse_poly_3_train)
print("R2:", r2_poly_3_train)
print("\nTesting Set:")
print("MSE:", mse_poly_3)
print("R2:", r2_poly_3)
```

```
Polynomial Model Performance:
Training Set:
MSE: 8.64661284180024
R2: 0.9870443297925774

Testing Set:
MSE: 9.034015244301722
R2: 0.9865149052434146
```

4.0.8 Putting all together

```python
# create a dataframe to put these 3 models together, including model name, features, training
models = ['Baseline', 'Polynomial Degree 2', 'Polynomial Degree 3']
features = [X.shape[1], len(poly_2.get_feature_names_out()), len(poly_3.get_feature_names_ou
training_mse = [mse_train_baseline, mse_train_poly_2, mse_poly_3_train]
testing_mse = [mse_baseline, mse_poly_2, mse_poly_3]
```

```python
training_r2 = [r2_train_baseline, r2_train_poly_2, r2_poly_3_train]
testing_r2 = [r2_baseline, r2_poly_2, r2_poly_3]

model_comparison = pd.DataFrame({
    'Model': models,
    'Features': features,
    'Training MSE': training_mse,
    'Testing MSE': testing_mse,
    'Training R2': training_r2,
    'Testing R2': testing_r2
})

model_comparison
```

	Model	Features	Training MSE	Testing MSE	Training R2	Testing R2
0	Baseline	2	218.849926	218.052227	0.672086	0.674513
1	Polynomial Degree 2	5	95.313782	90.001394	0.857186	0.865655
2	Polynomial Degree 3	9	8.646613	9.034015	0.987044	0.986515

```python
# print out the feature names for the polynomial degree 3 model
print("\nTransformed Feature Names:")
print(poly_3.get_feature_names_out())
```

```
Transformed Feature Names:
['x0' 'x1' 'x0^2' 'x0 x1' 'x1^2' 'x0^3' 'x0^2 x1' 'x0 x1^2' 'x1^3']
9
```

```python
# print out the feature names for the polynomial degree 2 model
print("\nTransformed Feature Names:")
print(poly_2.get_feature_names_out())
```

```
Transformed Feature Names:
['x0' 'x1' 'x0^2' 'x0 x1' 'x1^2']
5
```

4.0.9 degreee = 4

```
# use polynominal degree of 4 to see if it improves the model
poly_4 = PolynomialFeatures(degree=4, include_bias=False)

# Transform the training and testing features
X_train_poly_4 = poly_4.fit_transform(X_train)
X_test_poly_4 = poly_4.transform(X_test)

# Create a linear regression model for the polynomial features
poly_4_model = LinearRegression()

# Train the model on the transformed features
poly_4_model.fit(X_train_poly_4, y_train)

# Make predictions on the test set
y_pred_poly_4 = poly_4_model.predict(X_test_poly_4)

# Make predictions on the training set
y_pred_train_poly_4 = poly_4_model.predict(X_train_poly_4)

# Evaluate the polynomial model
mse_poly_4 = mean_squared_error(y_test, y_pred_poly_4)
r2_poly_4 = r2_score(y_test, y_pred_poly_4)

# Evaluate the polynomial model on the training set
mse_poly_4_train = mean_squared_error(y_train, y_pred_train_poly_4)
r2_poly_4_train = r2_score(y_train, y_pred_train_poly_4)

print("\nPolynomial Model Performance:")
print("Training Set:")
print("MSE:", mse_poly_4_train)
print("R2:", r2_poly_4_train)
print("\nTesting Set:")
print("MSE:", mse_poly_4)
print("R2:", r2_poly_4)
```

```
Polynomial Model Performance:
Training Set:
MSE: 8.633776015235346
```

```
R2: 0.98706356387817

Testing Set:
MSE: 8.991410070128255
R2: 0.9865785020821749
```

```
# get the feature names for the polynomial degree 4 model
print("\nNumber of Features:", len(poly_4.get_feature_names_out()))
print("\nTransformed Feature Names:")
print(poly_4.get_feature_names_out())
```

```
Number of Features: 14

Transformed Feature Names:
['x0' 'x1' 'x0^2' 'x0 x1' 'x1^2' 'x0^3' 'x0^2 x1' 'x0 x1^2' 'x1^3' 'x0^4'
 'x0^3 x1' 'x0^2 x1^2' 'x0 x1^3' 'x1^4']
```

4.1 Key takeaway:

In `scikit-learn`, the built-in `PolynomialFeatures` transformer is somewhat "all or nothing": by default, it generates **all** polynomial terms (including interactions) up to a certain degree. You can toggle:

- `interaction_only=True` to generate only cross-terms
- `include_bias=False` to exclude the constant (bias) term,
- `degree` to control how high the polynomial powers go.

However, if you want **fine-grained control** over exactly which terms get generated (for example, only certain interaction terms, or only a subset of polynomial terms), you will need to create those features manually or write a custom transformer (skipped for beginner level)

Use `interaction_only` for Cross Terms Only

If your goal is only to capture interaction terms (i.e., $ x_1 ×x_2 $, but no squares, cubes, etc.), you can set:

```
poly_int = PolynomialFeatures(degree=6,
                              interaction_only=True,
                              include_bias=False)

X_transformed = poly_int.fit_transform(X)
```

```
print("\nTransformed Feature Names:")
print(poly_int.get_feature_names_out())
```

```
Transformed Feature Names:
['x0' 'x1' 'x0 x1']
```

If you want to be very selective—say, just add x_1^2 and \$ x_1 ×x_2 \$ but not x_2^2 —the simplest approach is to create columns by hand. For example:

```
import numpy as np

X1 = X[:, 0].reshape(-1, 1)  # feature 1
X2 = X[:, 1].reshape(-1, 1)  # feature 2

# Manually create specific transformations
X1_sq = X1**2
X1X2 = X1 * X2

# Combine them as you like
X_new = np.hstack([X1, X2, X1_sq, X1X2])

print("\nTransformed Feature Names:")
print(['x1', 'x2', 'x1^2', 'x1*x2'])

X_new[:5]  # Display the first 5 rows
```

```
Transformed Feature Names:
['x1', 'x2', 'x1^2', 'x1*x2']

array([[ -1.25459881,  -1.0636448 ,   1.57401818,   1.3344475 ],
       [  4.50714306,  -0.26564341,  20.3143386 ,  -1.19729284],
       [  2.31993942,   3.54547393,   5.3821189 ,   8.22528473],
       [  0.98658484,  -1.59995614,   0.97334965,  -1.57849247],
       [ -3.4398136 ,   3.69649685,  11.83231757, -12.71526011]])
```

When using `PolynomialFeatures` (or any other scikit-learn transformer), the fitting step is always done on the training data—not on the test data. This is a fundamental principle of machine learning pipelines: we do not use the test set for any part of model training (including feature encoding, feature generation, scaling, etc.).

5 Beyond Fit (implementation)

```
# Import necessary libraries
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import statsmodels.api as sm
import seaborn as sns
import matplotlib.pyplot as plt
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
# Load the Boston Housing dataset (for demonstration purposes)
df = pd.read_csv('datasets/Housing.csv')
df.head()
```

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airco
0	13300000	7420	4	2	3	yes	no	no	no	
1	12250000	8960	4	4	4	yes	no	no	no	
2	12250000	9960	3	2	2	yes	no	yes	no	
3	12215000	7500	4	2	2	yes	no	yes	no	
4	11410000	7420	4	1	2	yes	yes	yes	no	

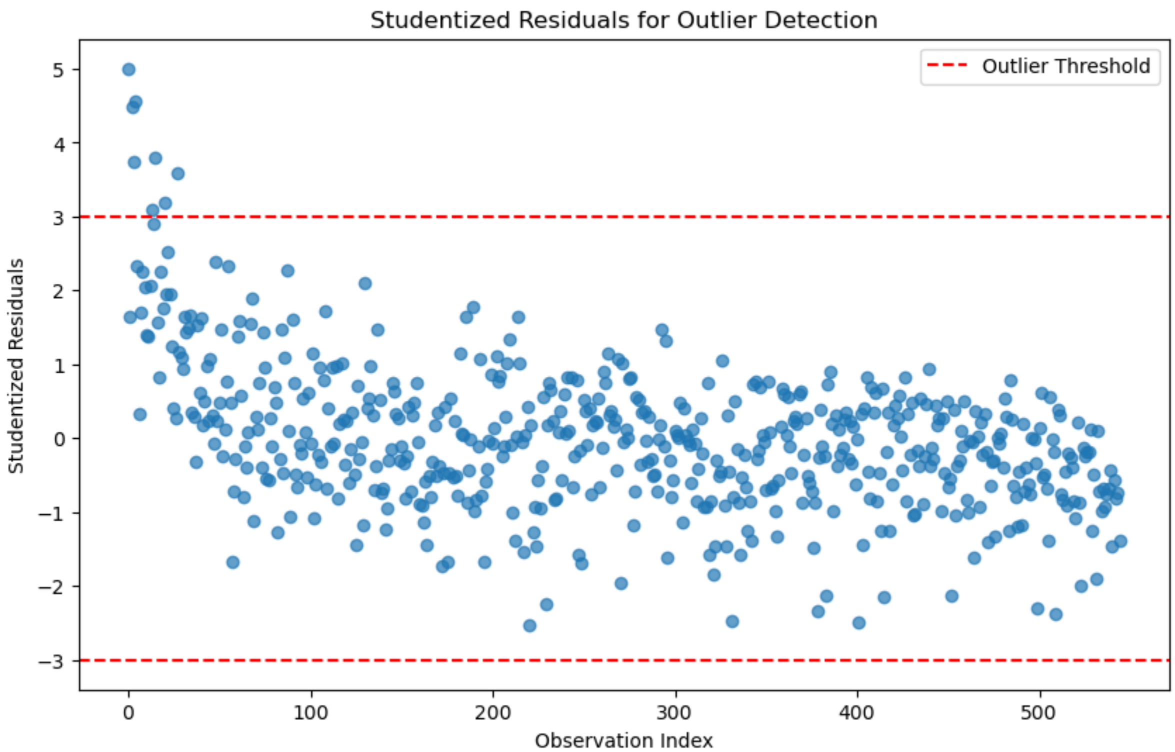
```
# build a formular api model using price as the target, the rest of the variables as predicto
model = smf.ols('price ~ area + bedrooms + bathrooms + stories + mainroad + guestroom + basem
model = model.fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.682
Model:                            OLS   Adj. R-squared:                  0.674
Method:                 Least Squares   F-statistic:                     87.52
Date:                Wed, 05 Feb 2025   Prob (F-statistic):          9.07e-123
```

```
Time:                        08:46:02   Log-Likelihood:                -8331.5
No. Observations:                 545   AIC:                         1.669e+04
Df Residuals:                     531   BIC:                         1.675e+04
Df Model:                          13
Covariance Type:            nonrobust
=====================================================================================================
                                        coef    std err          t      P>|t|      [0.025
-----------------------------------------------------------------------------------------------------
Intercept                          4.277e+04   2.64e+05      0.162      0.872   -4.76e+05
mainroad[T.yes]                    4.213e+05   1.42e+05      2.962      0.003    1.42e+05
guestroom[T.yes]                   3.005e+05   1.32e+05      2.282      0.023    4.18e+04
basement[T.yes]                    3.501e+05    1.1e+05      3.175      0.002    1.33e+05
hotwaterheating[T.yes]             8.554e+05   2.23e+05      3.833      0.000    4.17e+05
airconditioning[T.yes]              8.65e+05   1.08e+05      7.983      0.000    6.52e+05
prefarea[T.yes]                    6.515e+05   1.16e+05      5.632      0.000    4.24e+05
furnishingstatus[T.semi-furnished] -4.634e+04   1.17e+05     -0.398      0.691   -2.75e+05
furnishingstatus[T.unfurnished]    -4.112e+05   1.26e+05     -3.258      0.001   -6.59e+05
area                                244.1394     24.289     10.052      0.000     196.425
bedrooms                           1.148e+05   7.26e+04      1.581      0.114   -2.78e+04
bathrooms                          9.877e+05   1.03e+05      9.555      0.000    7.85e+05
stories                            4.508e+05   6.42e+04      7.026      0.000    3.25e+05
parking                            2.771e+05   5.85e+04      4.735      0.000    1.62e+05
==============================================================================
Omnibus:                       97.909   Durbin-Watson:                   1.209
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              258.281
Skew:                           0.895   Prob(JB):                     8.22e-57
Kurtosis:                       5.859   Cond. No.                     3.49e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.49e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

```python
# ------------------------------------------
# 1. Identifying Outliers (using studentized residuals)
# ------------------------------------------
# Outliers can be detected using studentized residuals
outliers_studentized = model.get_influence().resid_studentized_external
outlier_threshold = 3  # Common threshold for studentized residuals
```

```python
# Plot studentized residuals
plt.figure(figsize=(10, 6))
plt.scatter(range(len(outliers_studentized)), outliers_studentized, alpha=0.7)
plt.axhline(y=outlier_threshold, color='r', linestyle='--', label='Outlier Threshold')
plt.axhline(y=-outlier_threshold, color='r', linestyle='--')
plt.title('Studentized Residuals for Outlier Detection')
plt.xlabel('Observation Index')
plt.ylabel('Studentized Residuals')
plt.legend()
plt.show()
```




```python
# Identify observations with high studentized residuals
outlier_indices_studentized = np.where(np.abs(outliers_studentized) > outlier_threshold)[0]
print(f"Outliers detected at indices: {outlier_indices_studentized}")
```

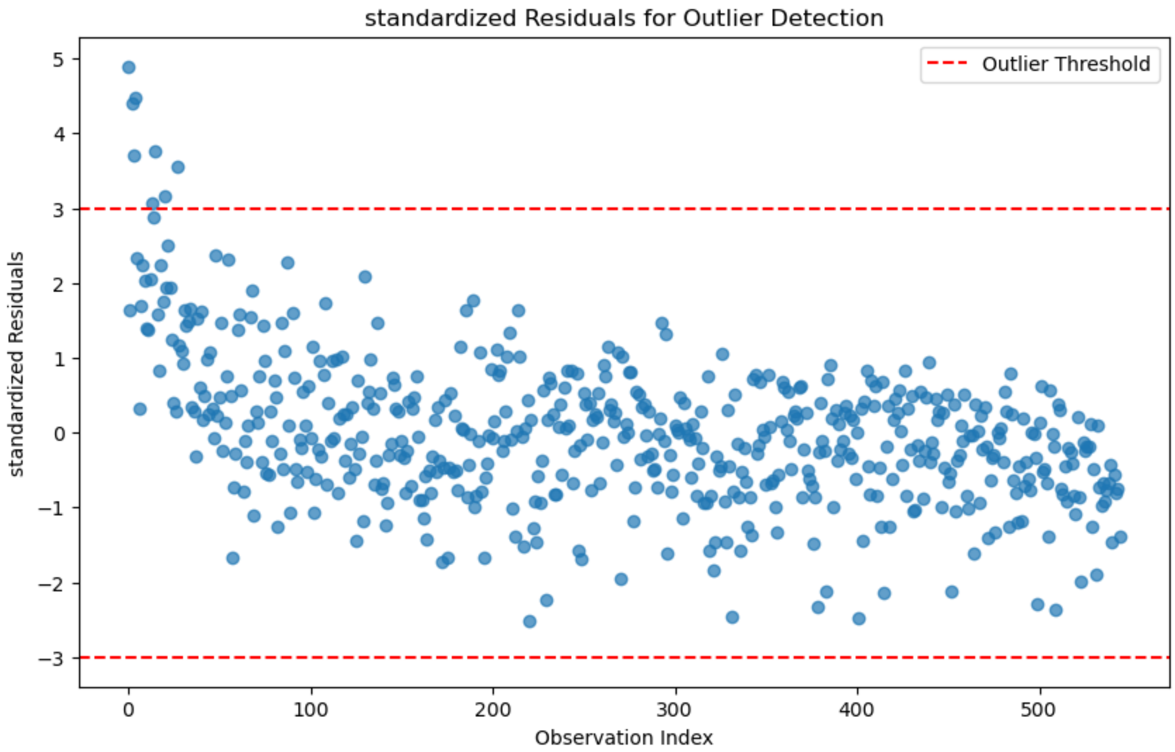
```
Outliers detected at indices: [ 0  2  3  4 13 15 20 27]
```

```python
# ------------------------------------------
# 1. Identifying Outliers (using standardized residuals)
# ------------------------------------------
# Outliers can be detected using standardized  residuals
outliers_standardized = model.get_influence().resid_studentized_internal
outlier_threshold = 3  # Common threshold for standardized residuals
```

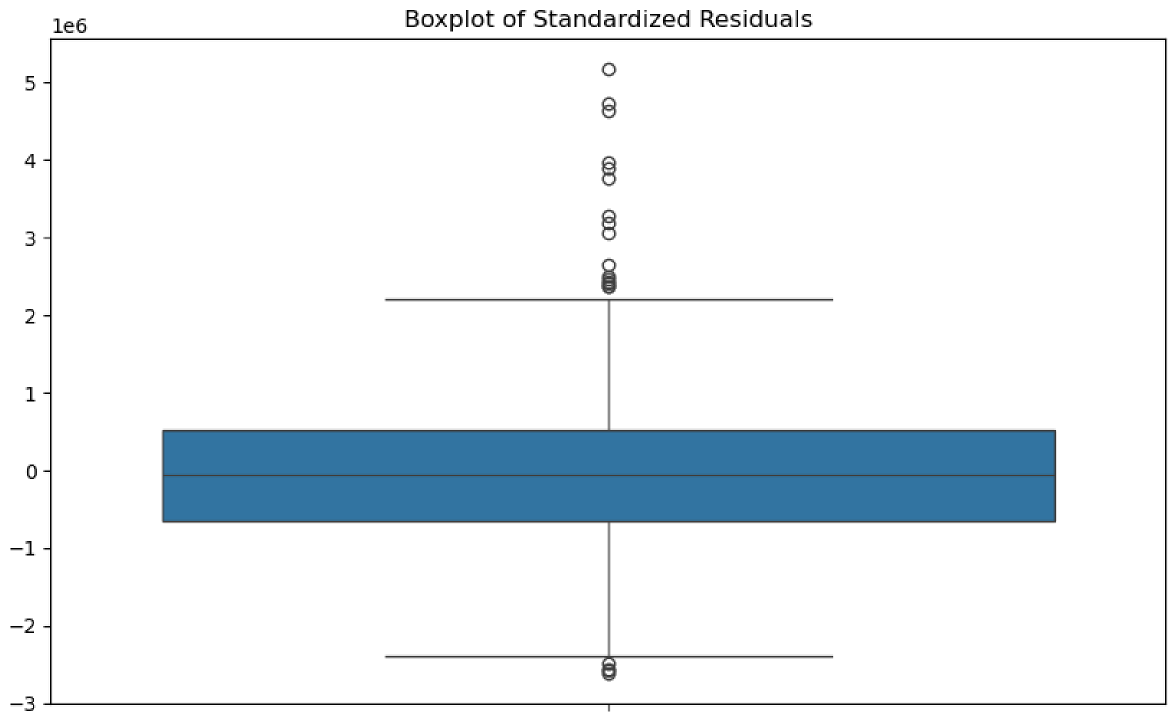
```python
# Identify observations with high standardized residuals
outlier_indices_standardized = np.where(np.abs(outliers_standardized) > outlier_threshold)[0]
print(f"Outliers detected at indices: {outlier_indices_standardized}")
```

```
Outliers detected at indices: [ 0  2  3  4 13 15 20 27]
```

```python
# Plot studentized residuals
plt.figure(figsize=(10, 6))
plt.scatter(range(len(outliers_standardized)), outliers_standardized, alpha=0.7)
plt.axhline(y=outlier_threshold, color='r', linestyle='--', label='Outlier Threshold')
plt.axhline(y=-outlier_threshold, color='r', linestyle='--')
plt.title('standardized Residuals for Outlier Detection')
plt.xlabel('Observation Index')
plt.ylabel('standardized Residuals')
plt.legend()
plt.show()
```




```
# ------------------------------------------
# 1. Identifying Outliers (using boxplot)
# ------------------------------------------
# Outliers can be detected using boxplot of standardized residuals
plt.figure(figsize=(10, 6))
sns.boxplot(model.resid)
plt.title('Boxplot of Standardized Residuals');
```




```
# use 3 standard deviation rule to identify outliers
outlier_indices = np.where(np.abs(model.resid) > 3 * model.resid.std())[0]
print(f"Outliers detected at indices: {outlier_indices}")
```

```
Outliers detected at indices: [ 0  2  3  4 13 15 20 27]
```

```
# ------------------------------------------
# 2. Identifying High Leverage Points
# ------------------------------------------
# High leverage points can be detected using the hat matrix (leverage values)
leverage = model.get_influence().hat_matrix_diag
leverage_threshold = 2 * (df.shape[1] / df.shape[0])  # Common threshold for leverage
```

5.0.0.1 Identifying High Leverage Points

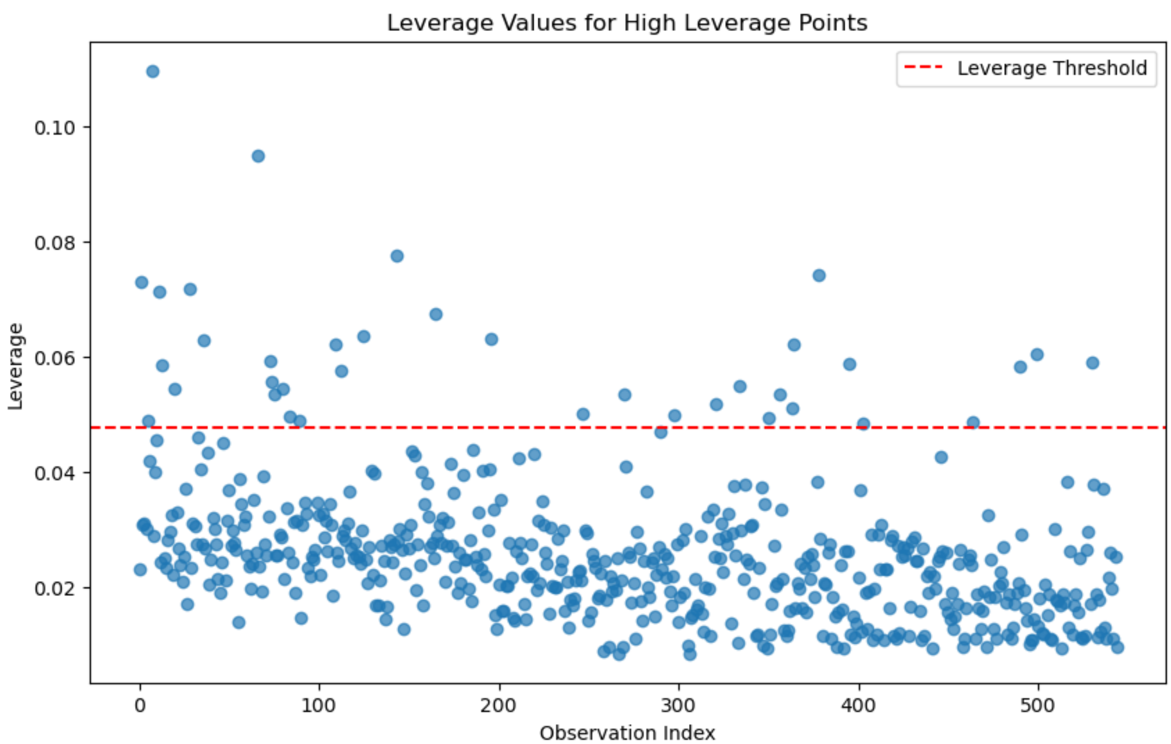
A common threshold for identifying **high leverage points** in regression analysis is:

$h_i > \frac{2p}{n}$

where:
- h_i is the leverage value for the (i)-th observation,

- p is the number of predictors (including the intercept), and
- n is the total number of observations.

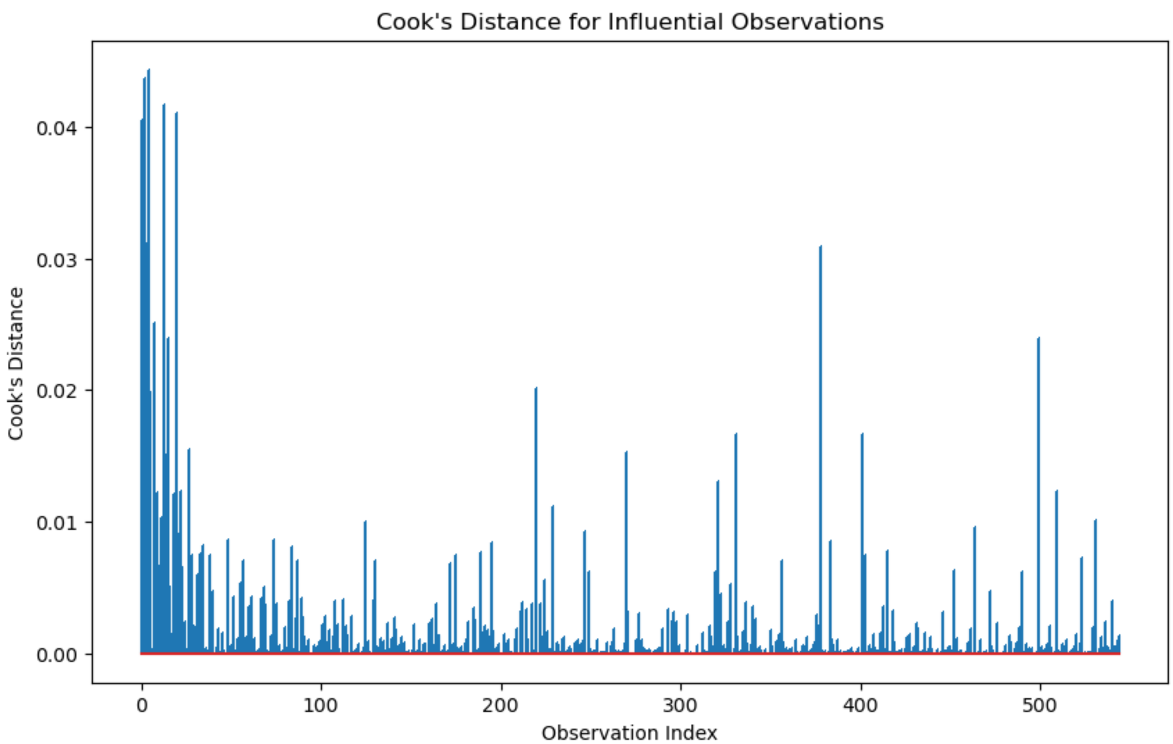
```
# Plot leverage values
plt.figure(figsize=(10, 6))
plt.scatter(range(len(leverage)), leverage, alpha=0.7)
plt.axhline(y=leverage_threshold, color='r', linestyle='--', label='Leverage Threshold')
plt.title('Leverage Values for High Leverage Points')
plt.xlabel('Observation Index')
plt.ylabel('Leverage')
plt.legend()
plt.show()
```




```
# Identify observations with high leverage
high_leverage_indices = np.where(leverage > leverage_threshold)[0]
print(f"High leverage points detected at indices: {high_leverage_indices}")
```

```
High leverage points detected at indices: [  1   5   7  11  13  20  28  36  66  73  74  75  8
 143 165 196 247 270 298 321 334 350 356 363 364 378 395 403 464 490 499
 530]
```

```
# ------------------------------------------
# 3. Cook's Distance for Influential Observations
# ------------------------------------------
# Cook's distance measures the influence of each observation on the model
cooks_distance = model.get_influence().cooks_distance[0]

# Plot Cook's distance
plt.figure(figsize=(10, 6))
plt.stem(range(len(cooks_distance)), cooks_distance, markerfmt=",")
plt.title("Cook's Distance for Influential Observations")
plt.xlabel('Observation Index')
plt.ylabel("Cook's Distance")
plt.show()
```



Cook's distance is considered high if it is greater than 0.5 and extreme if it is greater than 1.

```
# Identify influential observations
influential_threshold = 4 / (df.shape[1] - 1 ) # Common threshold for Cook's distance
influential_indices = np.where(cooks_distance > influential_threshold)[0]
print(f"Influential observations detected at indices: {influential_indices}")
```

```
Influential observations detected at indices: []
```

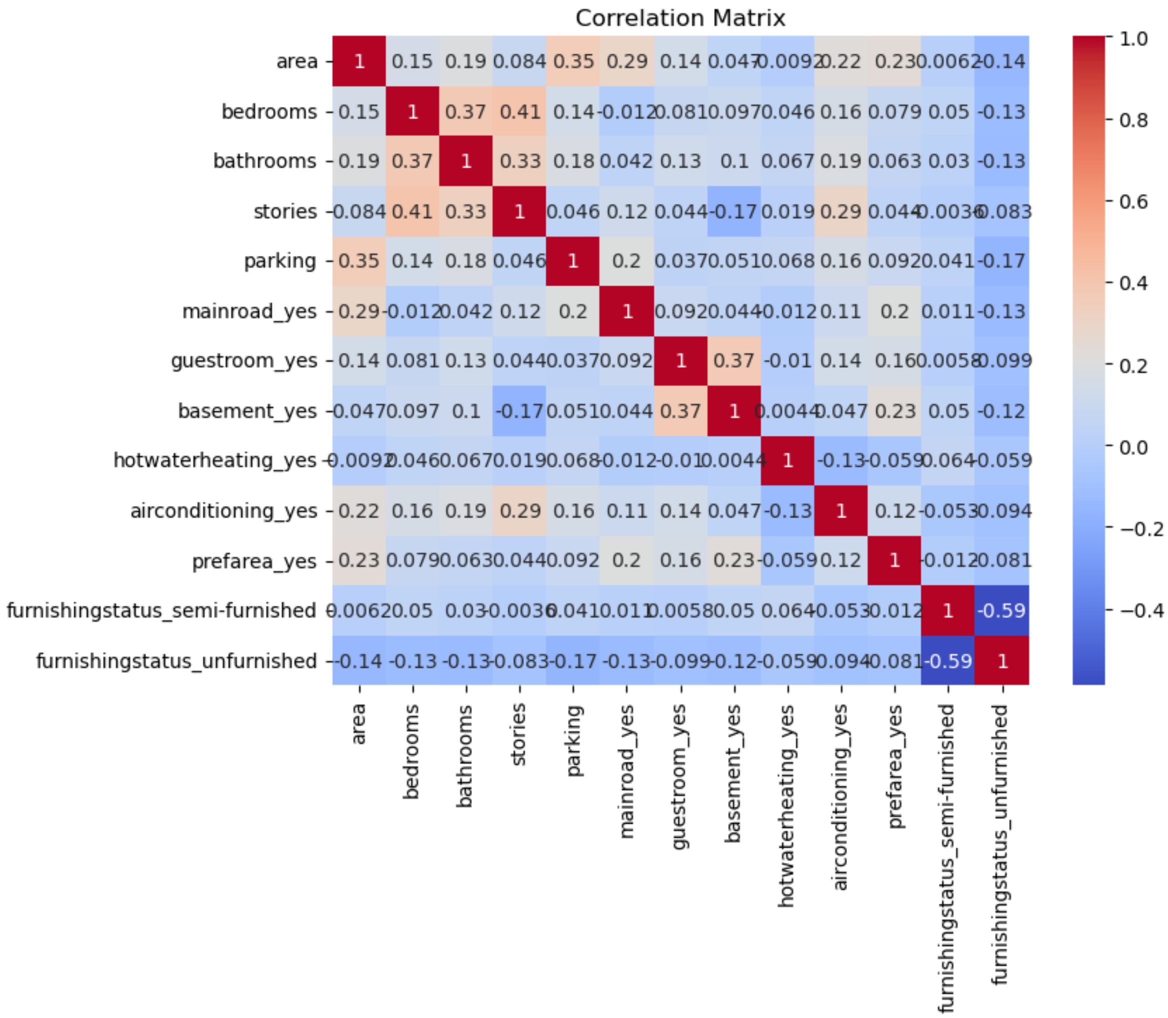
```python
# ========================================
# 4. Checking Multicollinearity (VIF)
# ========================================
# VIF calculation
from statsmodels.stats.outliers_influence import variance_inflation_factor

def calculate_vif(X):
    vif_data = pd.DataFrame()
    vif_data["Variable"] = X.columns
    vif_data["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vif_data

X = df[['area', 'bedrooms', 'bathrooms', 'stories', 'mainroad', 'guestroom', 'basement', 'ho

# one-hot encoding for categorical variables
X = pd.get_dummies(X, drop_first=True, dtype=float)


vif_data = calculate_vif(X)
print("\nVariance Inflation Factors:")
print(vif_data.sort_values('VIF', ascending=False))
```

```
Variance Inflation Factors:
                          Variable        VIF
1                         bedrooms  16.652387
2                        bathrooms   9.417643
0                             area   8.276447
3                          stories   7.880730
5                     mainroad_yes   6.884806
11  furnishingstatus_semi-furnished   2.386831
7                     basement_yes   2.019858
12     furnishingstatus_unfurnished   2.008632
4                          parking   1.986400
9              airconditioning_yes   1.767753
10                    prefarea_yes   1.494211
6                    guestroom_yes   1.473234
8              hotwaterheating_yes   1.091568
```

```python
# Rule of thumb: VIF > 10 indicates significant multicollinearity
multicollinear_features = vif_data[vif_data['VIF'] > 10]['Variable']
print(f"Features with significant multicollinearity: {multicollinear_features.tolist()}")
```

```
Features with significant multicollinearity: ['bedrooms']
```

```python
# ========================================
# 4. Checking Multicollinearity (Correlation Matrix)
# ========================================
# Correlation matrix
correlation_matrix = X.corr()
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix');
```

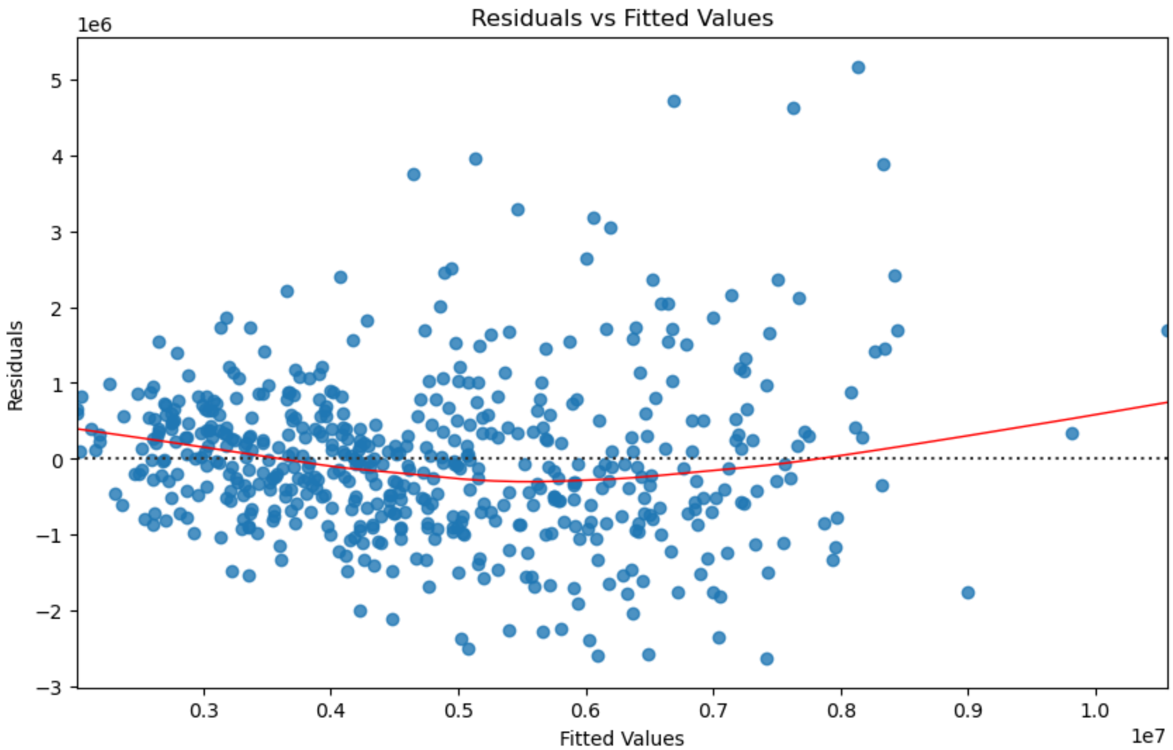



```
#output the correlation of other predictors with the bedrooms
X.corr()['bedrooms'].abs().sort_values(ascending=False)
```

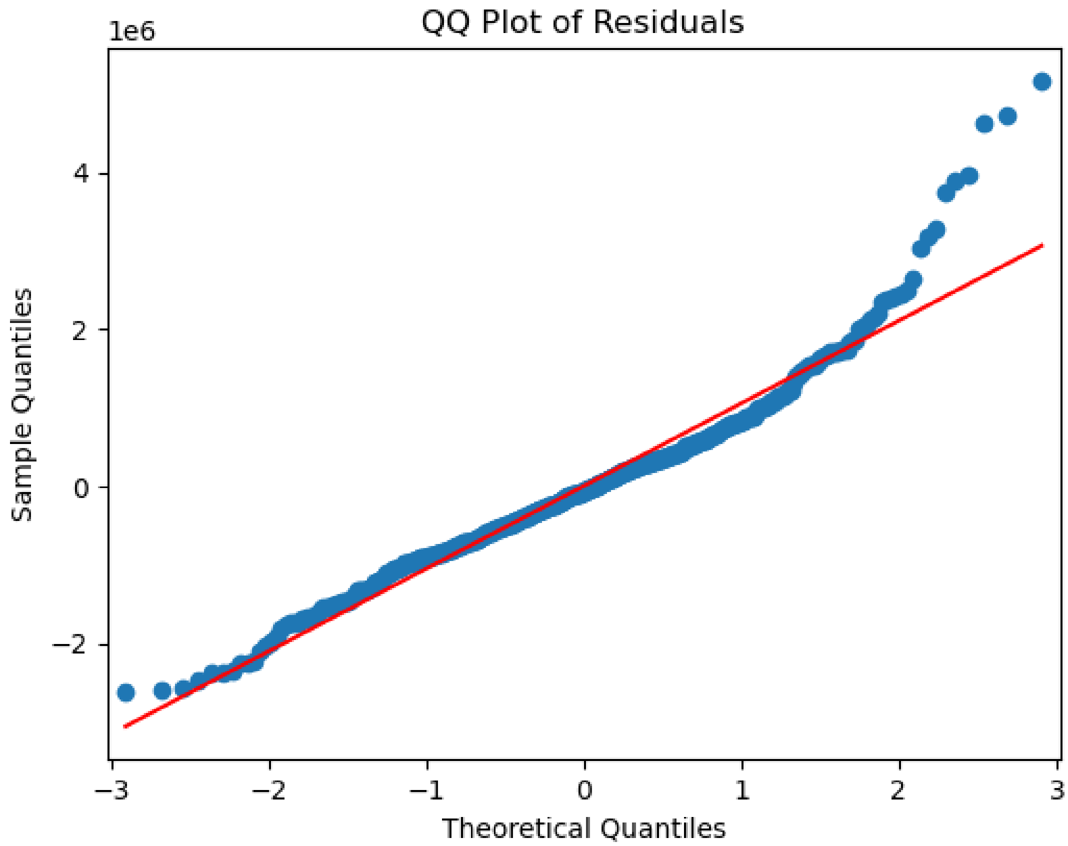
```
bedrooms                           1.000000
stories                            0.408564
bathrooms                          0.373930
airconditioning_yes                0.160603
area                               0.151858
parking                            0.139270
furnishingstatus_unfurnished       0.126252
basement_yes                       0.097312
guestroom_yes                      0.080549
prefarea_yes                       0.079023
```

```
furnishingstatus_semi-furnished    0.050040
hotwaterheating_yes                0.046049
mainroad_yes                       0.012033
Name: bedrooms, dtype: float64
```

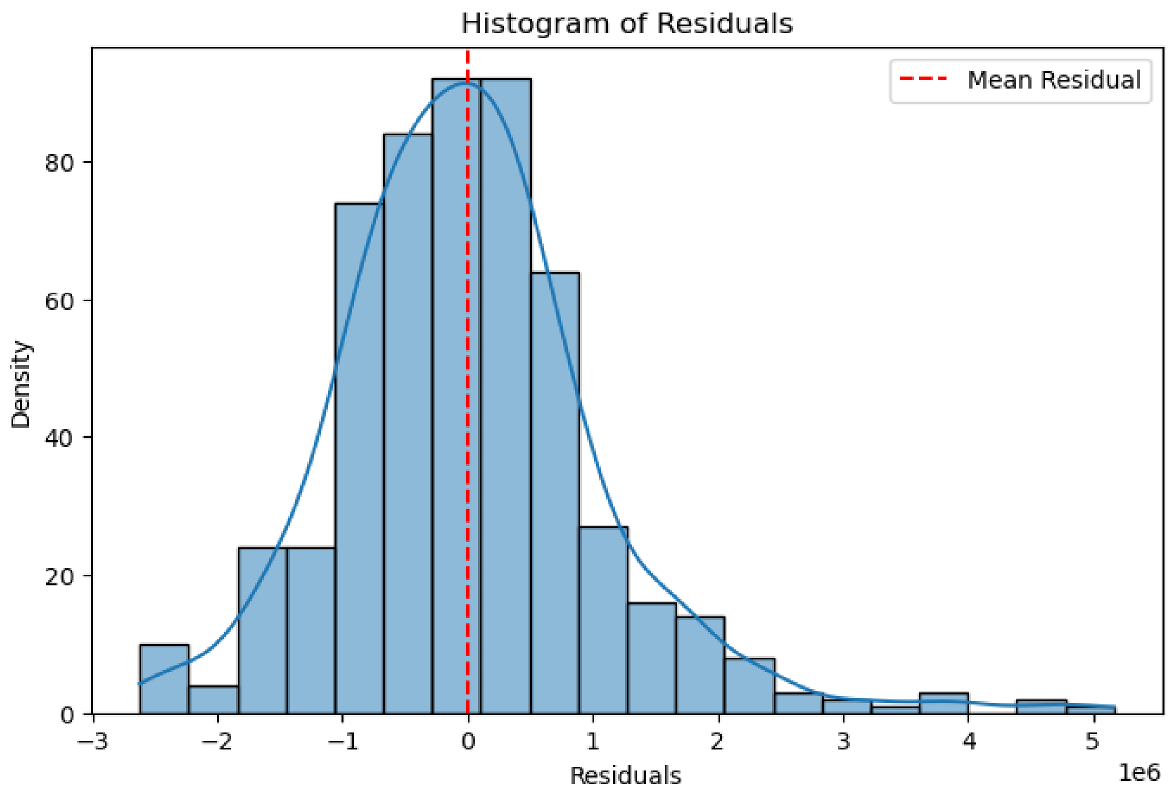
```
# -------------------------------------------
# 5. Analyzing Residual Patterns
# -------------------------------------------
# Residuals vs Fitted Values Plot
fitted_values = model.fittedvalues
residuals = model.resid

plt.figure(figsize=(10, 6))
sns.residplot(x=fitted_values, y=residuals, lowess=True, line_kws={'color': 'red', 'lw': 1})
plt.title('Residuals vs Fitted Values')
plt.xlabel('Fitted Values')
plt.ylabel('Residuals');

# QQ Plot for Normality of Residuals
plt.figure(figsize=(10, 6))
sm.qqplot(residuals, line='s')
plt.title('QQ Plot of Residuals');
```



<Figure size 1000x600 with 0 Axes>




```
plt.figure(figsize=(8, 5))
sns.histplot(residuals, kde=True, bins=20)
plt.axvline(residuals.mean(), color='red', linestyle='--', label="Mean Residual")
plt.xlabel("Residuals")
plt.ylabel("Density")
plt.title("Histogram of Residuals")
plt.legend()
plt.show()
```



Histogram of Residuals
Mean Residual
Density
0
20
40
60
80
−3
−2
−1
0
1
2
3
4
5
Residuals
1e6

6 Beyond Fit (statistical theory)

Read section 3.3.3 (4, 5, & 6) of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

Let us continue with the car price prediction example from the previous chapter.

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.api as sm
from scipy import stats
from sklearn.model_selection import cross_val_predict
from patsy import dmatrices
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

```
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
# Considering the model developed to address assumptions in the previous chapter
# Model with an interaction term and a variable transformation term
ols_object = smf.ols(formula = 'np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)
model_log = ols_object.fit()
model_log.summary()
```

Dep. Variable:	np.log(price)	**R-squared:**	0.803
Model:	OLS	**Adj. R-squared:**	0.803
Method:	Least Squares	**F-statistic:**	1834.
Date:	Sun, 10 Mar 2024	**Prob (F-statistic):**	0.00
Time:	16:51:01	**Log-Likelihood:**	-1173.8
No. Observations:	4960	**AIC:**	2372.
Df Residuals:	4948	**BIC:**	2450.
Df Model:	11		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	-238.2125	25.790	-9.237	0.000	-288.773	-187.652
year	0.1227	0.013	9.608	0.000	0.098	0.148
engineSize	13.8349	5.795	2.387	0.017	2.475	25.195
mileage	0.0005	0.000	3.837	0.000	0.000	0.001
mpg	-1.2446	0.345	-3.610	0.000	-1.921	-0.569
year:engineSize	-0.0067	0.003	-2.324	0.020	-0.012	-0.001
year:mileage	-2.67e-07	6.8e-08	-3.923	0.000	-4e-07	-1.34e-07
year:mpg	0.0006	0.000	3.591	0.000	0.000	0.001
engineSize:mileage	-2.668e-07	4.08e-07	-0.654	0.513	-1.07e-06	5.33e-07
engineSize:mpg	0.0028	0.000	6.842	0.000	0.002	0.004
mileage:mpg	7.235e-08	1.79e-08	4.036	0.000	3.72e-08	1.08e-07
I(mileage ** 2)	1.828e-11	5.64e-12	3.240	0.001	7.22e-12	2.93e-11

Omnibus:	711.514	**Durbin-Watson:**	0.498
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	2545.807
Skew:	0.699	**Prob(JB):**	0.00
Kurtosis:	6.220	**Cond. No.**	1.73e+13

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.73e+13. This might indicate that there are
strong multicollinearity or other numerical problems.

```
#Computing RMSE on test data
pred_price_log = model_log.predict(testf)
np.sqrt(((testp.price - np.exp(pred_price_log))**2).mean())
```

```
9094.209473232966
```

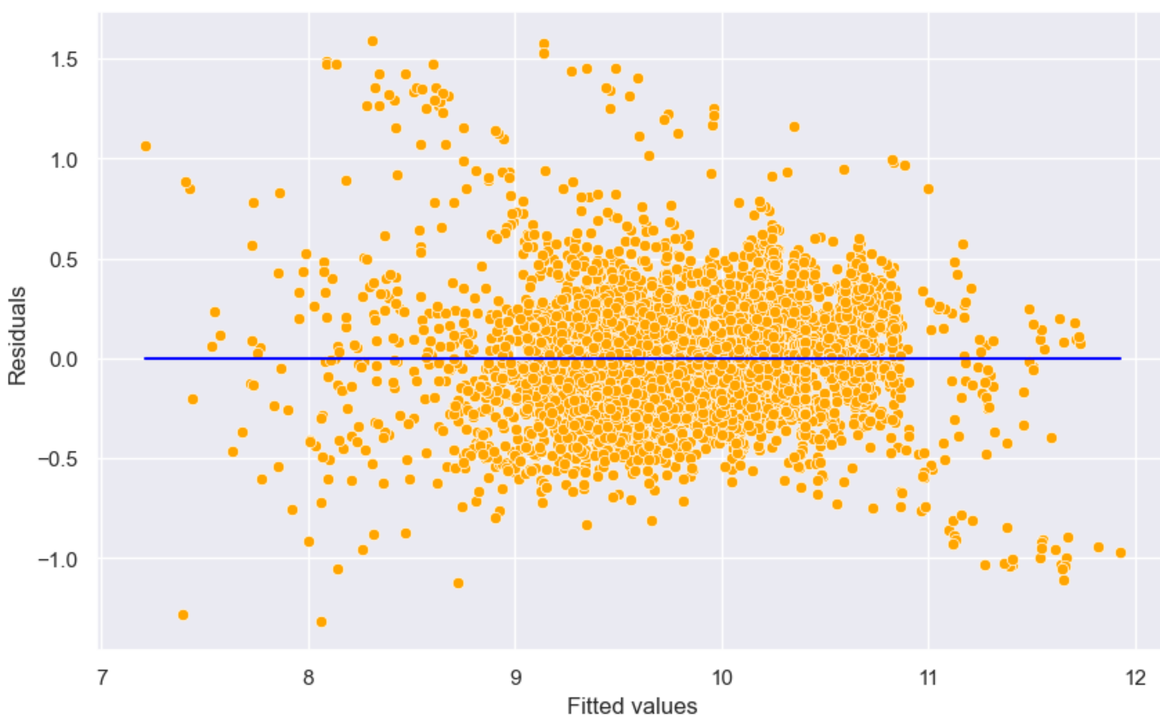
6.1 Outliers

An outlier is a point for which the true response (y_i) is far from the value predicted by the model. Residual plots can be used to identify outliers.

If the the response at the i^{th} observation is y_i, the prediction is $\hat{y}_i$, then the residual e_i is:

$$e_i = y_i - \hat{y}_i$$

```
#Plotting residuals vs fitted values
sns.set(rc={'figure.figsize':(10,6)})
sns.scatterplot(x = (model_log.fittedvalues), y=(model_log.resid),color = 'orange')
sns.lineplot(x = [model_log.fittedvalues.min(),model_log.fittedvalues.max()],y = [0,0],color
plt.xlabel('Fitted values')
plt.ylabel('Residuals');
```



Some of the errors may be high. However, it is difficult to decide how large a residual needs to be before we can consider a point to be an outlier. To address this problem, we have standardized residuals, which are defined as:

$$r_i = \frac{e_i}{RSE(\sqrt{1 - h_{ii}})},$$

where r_i is the standardized residual, RSE is the residual standard error, and h_{ii} is the leverage *(introduced in the next section)* of the i^{th} observation.

Standardized residuals, allow the residuals to be compared on a *standard scale*.

Issue with standardized residuals:, If the observation corresponding to the standardized residual has a high leverage, then it will drag the regression line / plane / hyperplane towards it, thereby influencing the estimate of the residual itself.

Studentized residuals: To address the issue with standardized residuals, studentized residual for the i^{th} observation is computed as the standardized residual, but with the RSE (residual standard error) computed after removing the i^{th} observation from the data. Studentized residual, t_i for the i^{th} observation is given as:

$$t_i = \frac{e_i}{RSE_i(\sqrt{1 - h_{ii}})},$$

where RSE_i is the residual standard error of the model developed on the data without the i^{th} observation.

Distribution of studentized residuals: If the regression model is appropriate such that no case is outlying because of a change in the model, then each studentized residual will follow a t distribution with (n–p–1) degrees of freedom.

As the studentized residuals follow a t distribution, we can conduct a hypothesis test to identify whether an observation is an outlier or not for a given sigificance level. Note that the test will be two-sided since we are not concerned with the sign of the residuals, but only their absolute values.

In the current example, for a signficance level of 5%, the critical t-statistic is $t(1 - \frac{\alpha}{2}, n - p - 1)$, as calculated below.

```
n = train.shape[0]
p = model_log.df_model
alpha = 0.05

# Critical value
stats.t.ppf(1 - alpha/2, n - p - 1)
```

```
1.9604435402730618
```

If we were conducting the test for a single observation, we'll compare the studentized residual for that observation with the critical t-statistic, and if the residual is greater than the critical value, we'll consider that observation as an outlier.

However, typically, we'll be interested in conducting this test for all observations, and thus we'll need a more conservative critical value for the same signficance level. This critical value is given by the Bonferroni correction as $t(1-\frac{\alpha}{2n}, n-p-1)$.

Thus, the minimum value of studentized residual for which the observation will be classified as an outlier is:

```
critical_value = stats.t.ppf(1-alpha/(2*n), n - p - 1)
critical_value
```

```
4.4200129981725365
```

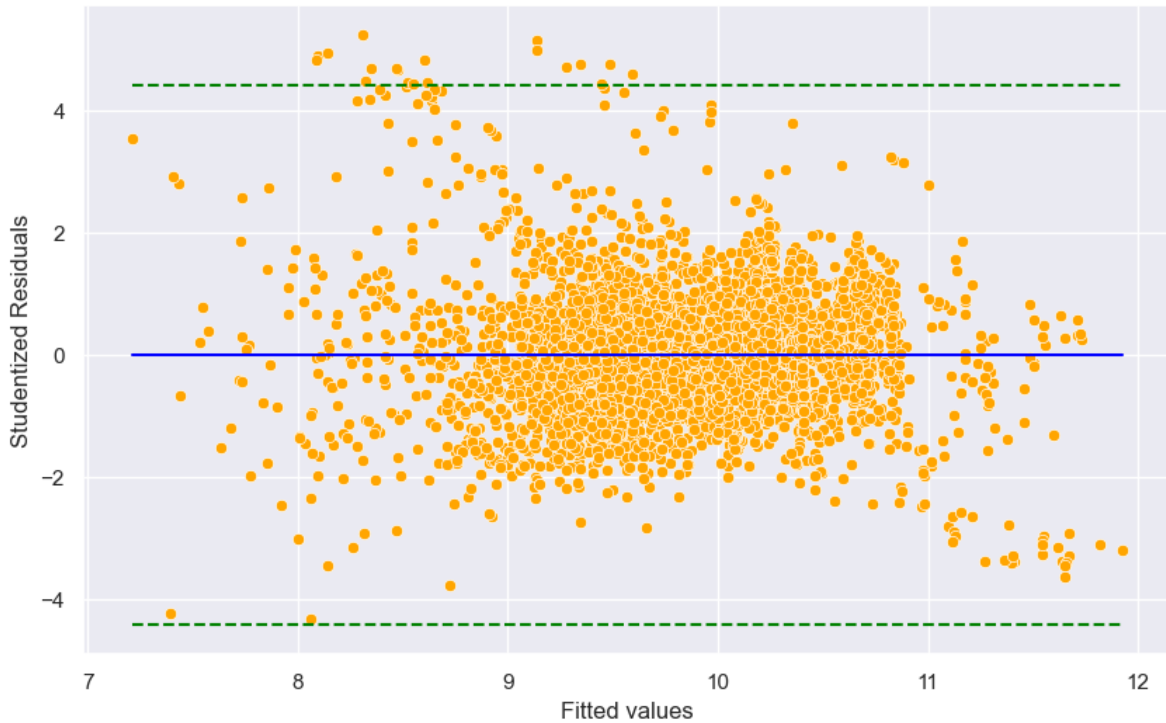
The studentized residuals can be obtained using the `outlier_test()` method of the object returned by the `fit()` method of an OLS object. Let us find the studentized residuals in our car `price` prediction model.

```
#Studentized residuals
out = model_log.outlier_test()
out
```

	student_resid	unadj_p	bonf(p)
0	-1.164204	0.244398	1.0
1	-0.801879	0.422661	1.0
2	-1.263820	0.206354	1.0
3	-0.614171	0.539131	1.0
4	0.027929	0.977720	1.0
…	…	…	…
4955	-0.523361	0.600747	1.0
4956	-0.509538	0.610398	1.0
4957	-1.718808	0.085712	1.0
4958	-0.077594	0.938154	1.0
4959	-0.482388	0.629551	1.0

Studentized residuals are in the first column of the above table. Let us plot the studentized residuals against fitted values. In the figure below, the studentized residuals above the top dotted green line and below the bottom dotted green line are outliers.

```
#Plotting studentized residuals vs fitted values
sns.scatterplot(x = (model_log.fittedvalues), y=(out.student_resid),color = 'orange')
sns.lineplot(x = [model_log.fittedvalues.min(),model_log.fittedvalues.max()],y = [0,0],color
ax = sns.lineplot(x = [model_log.fittedvalues.min(),model_log.fittedvalues.max()],y = [critic
             color = 'green')
sns.lineplot(x = [model_log.fittedvalues.min(),model_log.fittedvalues.max()],y = [-critical_v
             color = 'green')
ax.lines[1].set_linestyle("--")
ax.lines[2].set_linestyle("--")

plt.xlabel('Fitted values')
plt.ylabel('Studentized Residuals');
```



Outliers: Observations whose studentized residuals have a magnitude greater than $t(1 - \frac{\alpha}{2n}, n - p - 1)$.

Impact of outliers: Outliers do not have a large impact on the OLS line / plane / hyperplane as long as they don't have a high leverage *(discussed in the next section)*. However, outliers do inflate the residual standard error (RSE). RSE in turn is used to compute the standard errors of regression coefficients. As a result, statistically significant variables may appear to be insignificant, and R^2 may appear to be lower.

Are there outliers in our example?

```
#Number of points with absolute studentized residuals greater than critical_value
np.sum(np.abs(out.student_resid) > critical_value)
```

```
19
```

Let us analyze the outliers.

```
ind = (np.abs(out.student_resid) > critical_value)
pd.concat([train.loc[ind,:], np.exp(model_log.fittedvalues[ind])], axis = 1)
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price	0
2042	18228	bmw	i3	2017	Automatic	24041	Hybrid	0	78.2726	0.0	2149	
2046	17362	bmw	i3	2016	Automatic	68000	Hybrid	0	78.0258	0.0	1599	
2050	19224	bmw	i3	2016	Automatic	20013	Hybrid	0	77.9310	0.0	1987	
2051	13913	bmw	i3	2014	Automatic	34539	Hybrid	0	78.3838	0.0	1449	
2055	16512	bmw	i3	2017	Automatic	28169	Hybrid	0	77.9799	0.0	2375	
2059	15844	bmw	i3	2016	Automatic	19995	Hybrid	0	78.2825	0.0	1985	
2060	12107	bmw	i3	2016	Automatic	8421	Hybrid	0	77.9125	0.0	1949	
2061	18215	bmw	i3	2014	Automatic	37161	Hybrid	0	77.7505	0.0	1418	
2063	15617	bmw	i3	2017	Automatic	41949	Hybrid	140	78.1907	0.0	1999	
2064	18020	bmw	i3	2015	Automatic	9886	Hybrid	0	78.1810	0.0	1748	
2143	12972	bmw	i8	2017	Automatic	9992	Hybrid	135	69.2767	1.5	5995	
2144	13826	bmw	i8	2015	Automatic	43323	Hybrid	0	69.2683	1.5	4499	
2150	18949	bmw	i8	2015	Automatic	43102	Hybrid	0	69.0922	1.5	4289	
2151	18977	bmw	i8	2016	Automatic	10087	Hybrid	0	68.9279	1.5	4899	
2744	18866	merc	M Class	2004	Automatic	121000	Diesel	325	29.3713	2.7	1995	
3548	13149	audi	S4	2019	Automatic	4900	Diesel	145	40.7030	0.0	4500	
4116	16420	audi	SQ5	2020	Automatic	1500	Diesel	145	34.7968	0.0	5645	
4117	17611	audi	SQ5	2019	Automatic	1500	Diesel	145	34.5016	0.0	4880	
4851	16577	bmw	Z3	2002	Automatic	16500	Petrol	325	29.7614	2.2	1499	

Do you notice some unique characteristics of these observations due to which they may be outliers?

What methods you can propose to estimate the price of these outliers more accurately, which will also result in the overall reduction in RMSE?

6.2 High leverage points

High leverage points are those with an unsual value of the predictor(s). They have the potential to have a relatively higher impact on the OLS line / plane / hyperplane, as compared to the outliers.

Leverage statistic (page 99 of the book): In order to quantify an observation's leverage, we compute the leverage statistic. A large value of this statistic indicates an observation with high leverage. For simple linear regression,

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{i'=1}^{n}(x_{i'} - \bar{x})^2}. \tag{6.1}$$

It is clear from this equation that h_i increases with the distance of x_i from $\bar{x}$. A large value of h_i indicates that the i^{th} observation is distance from the center of all the other observations in terms of predictor values.

The leverage statistic h_i is always between $1/n$ and 1, and the average leverage for all the observations is always equal to $(p+1)/n$:

$$\bar{h} = \frac{p+1}{n} \tag{6.2}$$

So if a given observation has a leverage statistic that greatly exceeds $(p+1)/n$, then we may suspect that the corresponding point has high leverage.

If the i^{th} observation has a large leverage h_i, it may exercise substantial leverage in determining the fitted value $\widehat{Y}_i$, because:

- The fitted value $\widehat{Y}_i$ is a linear combination of the observed Y values, and h_i is the weight of observation Y_i in determining this fitted value.
- The larger the h_i, the smaller is the variance of the residual e_i, and the closer the fitted value $\widehat{Y}_i$ will tend to be the observed value Y_i.

Thumb rules:

- A leverage h_i is usually considered large if it is more than twice as large as the mean value $\bar{h}$.
- Another suggested guideline is that h_i values exceeding 0.5 indicate **very high leverage**, whereas those between 0.2 and 0.5 indicate moderate leverage.

Influential points: Note that if a high leverage point falls in line with the regression line, then it will not affect the regression line. However, it may inflate R-squared and increase the significance of predictors. If a high leverage point falls away from the regression line, then it is also an outlier, and will affect the regression line. The points whose presence significantly affects the regression line are called influential points. A point that is both a high leverage point and an outlier is likely to be an influential point. However, a high leverage point is not necessarily an influential point.

Source for influential points: https://online.stat.psu.edu/stat501/book/export/html/973

Let us see if there are any high leverage points in our regression `model`.

```
#Model with an interaction term and a variable transformation term
ols_object = smf.ols(formula = 'np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)
model_log = ols_object.fit()
model_log.summary()
```

Dep. Variable:	np.log(price)	**R-squared:**	0.803
Model:	OLS	**Adj. R-squared:**	0.803
Method:	Least Squares	**F-statistic:**	1834.
Date:	Sun, 10 Mar 2024	**Prob (F-statistic):**	0.00
Time:	16:53:39	**Log-Likelihood:**	-1173.8
No. Observations:	4960	**AIC:**	2372.
Df Residuals:	4948	**BIC:**	2450.
Df Model:	11		
Covariance Type:	nonrobust		

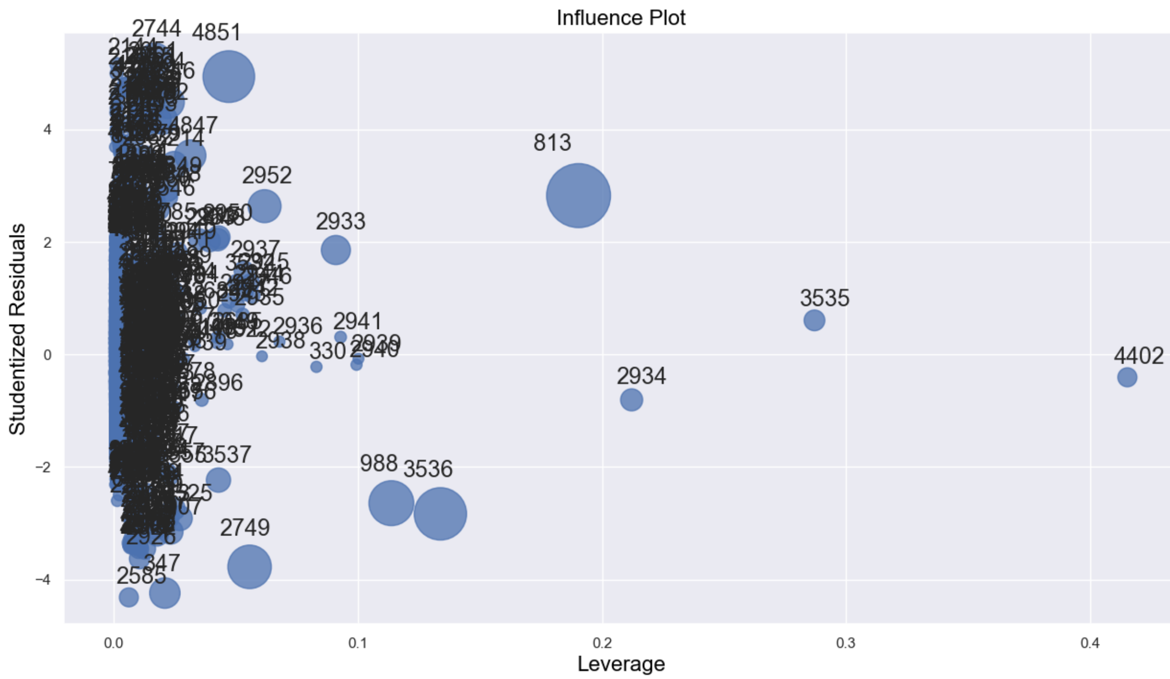
	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	-238.2125	25.790	-9.237	0.000	-288.773	-187.652
year	0.1227	0.013	9.608	0.000	0.098	0.148
engineSize	13.8349	5.795	2.387	0.017	2.475	25.195
mileage	0.0005	0.000	3.837	0.000	0.000	0.001
mpg	-1.2446	0.345	-3.610	0.000	-1.921	-0.569
year:engineSize	-0.0067	0.003	-2.324	0.020	-0.012	-0.001
year:mileage	-2.67e-07	6.8e-08	-3.923	0.000	-4e-07	-1.34e-07
year:mpg	0.0006	0.000	3.591	0.000	0.000	0.001
engineSize:mileage	-2.668e-07	4.08e-07	-0.654	0.513	-1.07e-06	5.33e-07
engineSize:mpg	0.0028	0.000	6.842	0.000	0.002	0.004
mileage:mpg	7.235e-08	1.79e-08	4.036	0.000	3.72e-08	1.08e-07
I(mileage ** 2)	1.828e-11	5.64e-12	3.240	0.001	7.22e-12	2.93e-11

Omnibus:	711.514	**Durbin-Watson:**	0.498
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	2545.807
Skew:	0.699	**Prob(JB):**	0.00
Kurtosis:	6.220	**Cond. No.**	1.73e+13

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.73e+13. This might indicate that there are
strong multicollinearity or other numerical problems.

```
#Computing the leverage statistic for each observation
influence = model_log.get_influence()
leverage = influence.hat_matrix_diag
```

```
#Visualizng leverage against studentized residuals
sns.set(rc={'figure.figsize':(15,8)})
sm.graphics.influence_plot(model_log);
```



Let us identify the high leverage points in the data, as they may be affecting the regression line if they are outliers as well, i.e., if they are influential points. Note that there is no defined threshold for a point to be classified as a high leverage point. Some statisticians consider points having twice the average leverage as high leverage points, some consider points having thrice the average leverage as high leverage points, and so on.

```
out = model_log.outlier_test()
```

```
#Average leverage of points
average_leverage = (model_log.df_model+1)/model_log.nobs
average_leverage
```

```
0.0024193548387096775
```

Let us consider points having four times the average leverage as high leverage points.

```
#We will remove all observations that have leverage higher than the threshold value.
high_leverage_threshold = 3*average_leverage
```

```
#Number of high leverage points in the dataset
np.sum(leverage>high_leverage_threshold)
```

```
269
```

6.2.1 Identifying extrapolation using leverage

Leverage can be used to check if prediction on a particular point will lead to extrapolation.

Below is the function that can be used to find the leverage at for a particular observation `xnew`. Note that `xnew` has to be a single-dimensional array, and `X` has to be the predictor matrix (also called the design matrix).

```
def leverage_compute(xnew, X):
    return(xnew.reshape(-1, 1).T.dot(np.linalg.inv(X.T.dot(X))).dot(xnew.reshape(-1, 1))[0][
```

As expected, the function will return the same leverage as provided by the `hat_matrix_diag` attribute of the objected returned by the `get_influence()` method of `model_log` as shown below:

```
leverage[0]
```

```
0.0026426981240353694
```

As the observation for prediction is required we need to create the predictor matrix `X` to create all the observations with the interactions specified in the model.

```
y, X = dmatrices('np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)', data = trai
```

```
leverage_compute(X[0,:], X)
```

```
0.0026426973869101977
```

If the leverage for a new observation is higher than the maximum leverage among all the observations in the training dataset, then prediction at the new observation will be extrapolation.

6.3 Influential points

Observations that are both high leverage points and outliers are influential points that may affect the regression line. Let's remove these influential points from the data and see if it improves the model prediction accuracy on test data.

```
#Dropping influential points from data
train_filtered = train.drop(np.intersect1d(np.where(np.abs(out.student_resid) > critical_valu
                                           (np.where(leverage>high_leverage_threshold)[0])))
```

Note that as the Bonferroni's critical value is very conservative estimate, we have rounded off the critical value to 4, instead of 4.42.

```
train_filtered.shape
```

```
(4948, 11)
```

```
#Number of points removed as they were influential
train.shape[0]-train_filtered.shape[0]
```

```
12
```

We removed 12 influential data points from the training data.

```
#Model after removing the influential observations
ols_object = smf.ols(formula = 'np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)
model_log = ols_object.fit()
model_log.summary()
```

Dep. Variable:	np.log(price)	**R-squared:**	0.815
Model:	OLS	**Adj. R-squared:**	0.814
Method:	Least Squares	**F-statistic:**	1971.
Date:	Sun, 10 Mar 2024	**Prob (F-statistic):**	0.00
Time:	16:54:08	**Log-Likelihood:**	-1027.9
No. Observations:	4948	**AIC:**	2080.
Df Residuals:	4936	**BIC:**	2158.
Df Model:	11		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	-256.2339	25.421	-10.080	0.000	-306.070	-206.398
year	0.1317	0.013	10.462	0.000	0.107	0.156
engineSize	18.4650	5.663	3.261	0.001	7.364	29.566
mileage	0.0006	0.000	4.288	0.000	0.000	0.001
mpg	-1.1810	0.338	-3.489	0.000	-1.845	-0.517
year:engineSize	-0.0090	0.003	-3.208	0.001	-0.015	-0.004
year:mileage	-2.933e-07	6.7e-08	-4.374	0.000	-4.25e-07	-1.62e-07
year:mpg	0.0006	0.000	3.458	0.001	0.000	0.001
engineSize:mileage	-4.316e-07	4e-07	-1.080	0.280	-1.21e-06	3.52e-07
engineSize:mpg	0.0048	0.000	11.537	0.000	0.004	0.006
mileage:mpg	7.254e-08	1.75e-08	4.140	0.000	3.82e-08	1.07e-07
I(mileage ** 2)	1.668e-11	5.53e-12	3.017	0.003	5.84e-12	2.75e-11

Omnibus:	718.619	**Durbin-Watson:**	0.521
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	2512.509
Skew:	0.714	**Prob(JB):**	0.00
Kurtosis:	6.185	**Cond. No.**	1.75e+13

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.75e+13. This might indicate that there are strong multicollinearity or other numerical problems.

Let us compare the square root of 5-fold cross-validated mean squared error of the model with and without the influential points.

```
y, X = dmatrices('np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)', data = trai
np.sqrt(mean_squared_error(np.exp(cross_val_predict(LinearRegression(), X, y)), np.exp(y)))
```

```
9811.74078331643
```

```
y, X = dmatrices('np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)', data = trai
np.sqrt(mean_squared_error(np.exp(cross_val_predict(LinearRegression(), X, y)), np.exp(y)))
```

```
9800.202063309154
```

Why can't we use `cross_val_score()` instead of `cross_val_predict()` here?

There seems to be a slight improvement in prediction error after removing influential points. Note that none of the points had "very high leverage", and thus the change is not substantial.

Note that we obtain a higher R-squared value of 81.5% as compared to 80% with the complete data. Removing the influential points helped obtain a slightly better model fit. However, that may also happen just by reducing observations.

```
#Computing RMSE on test data
pred_price_log = model_log.predict(testf)
np.sqrt(((testp.price - np.exp(pred_price_log))**2).mean())
```

```
8922.977452912108
```

The RMSE on test data has also reduced. This shows that some of the influential points were impacting the regression line. With those points removed, the model better captures the general trend in the data.

6.3.1 Influence on single fitted value (DFFITS)

- A useful measure of the influence that the i^{th} observation has on the fitted value $\hat{Y}_i$ is:

$$(DFFITS)_i = \frac{\hat{Y}_i - \hat{Y}_{i(i)}}{\sqrt{MSE_i h_i}} \tag{6.3}$$

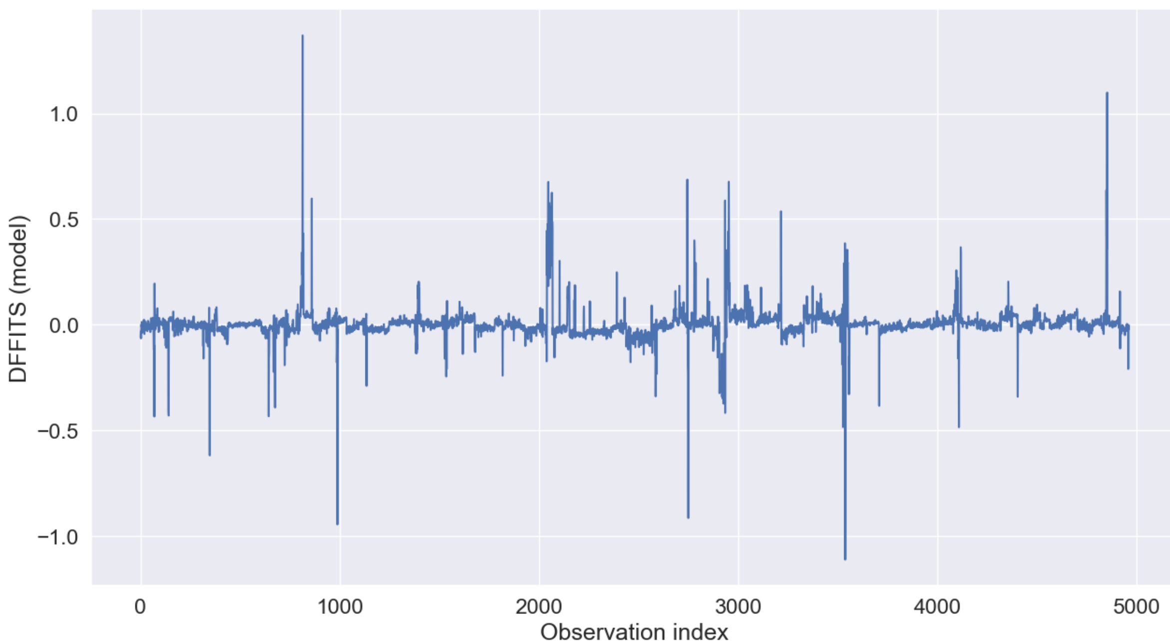
- Note that the denominator in the above fraction is the estimated standard deviation of $\hat{Y}_i$, but uses the error mean square when the i^{th} observation is omitted.
- $DFFITS$ for the i^{th} observation represents the number of estimated standard deviations of $\hat{Y}_i$ that the fitted value $\hat{Y}_i$ increases or decreases with the inclusion of the i^{th} observation in fitting the regression model.
- It can be shown that:

$$(DFFITS)_i = t_i \sqrt{\frac{h_i}{1 - h_i}} \tag{6.4}$$

where t_i is the studentized deleted residual for the i^{th} observation.

- We can see that if an observation has high leverage and is an outlier, it is likely to be influential
- For large datasets, an observation is considered influential if the magnitude of $DFFITS$ for it exceeds $2\sqrt{\frac{p}{n}}$

```
sns.set(font_scale =1.5)
sns.lineplot(x = range(train.shape[0]), y = influence.dffits[0])
plt.xlabel('Observation index')
plt.ylabel('DFFITS (model)');
```



Let us analyze the point with the highest $DFFITS$.

```
np.where(influence.dffits[0]>1)
```

```
(array([ 813, 4851], dtype=int64),)
```

```
train.loc[813,:]
```

```
carID                12454
brand                   vw
model            Caravelle
year                  2012
transmission     Semi-Auto
mileage             212000
fuelType            Diesel
tax                    325
mpg                34.4424
engineSize             2.0
price                11995
Name: 813, dtype: object
```

```
train.loc[train.model == ' Caravelle','mileage'].describe()
```

```
count        65.000000
mean      25638.692308
std       42954.135726
min          10.000000
25%        3252.000000
50%        6900.000000
75%       30414.000000
max      212000.000000
Name: mileage, dtype: float64
```

```
# Prediction with model developed based on all points
ols_object = smf.ols(formula = 'np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)
                     data = train)
model_log = ols_object.fit();
np.exp(model_log.predict(train.loc[[813],:]))
```

```
813    5502.647323
dtype: float64
```

```
# Prediction with model developed based on all points except the 813th point
ols_object = smf.ols(formula = 'np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**2)
                     data = train.drop(index = 813))
model_log = ols_object.fit();
np.exp(model_log.predict(train.loc[[813],:]))
```

```
813    4581.374593
dtype: float64
```

Let us see the leverage and studentized residual for this observation.

```
# Leverage
leverage[813]
```

```
0.19038697461006687
```

```
# Studentized residual
out.student_resid[813]
```

```
2.823478041409651
```

Do you notice what may be contributing to the high influence of this point?

6.3.2 Influence on all fitted values (Cook's distance)

In contrast to $DFFITS$, which considers the influence of the i^{th} observation on the fitted value $\hat{Y}_i$, Cook's distance considers the influence of the i^{th} observation on all n the fitted values:

$$D_i = \frac{\sum_{j=1}^{n}(\hat{Y}_j - \hat{Y}_{j(i)})^2}{pMSE} \tag{6.5}$$

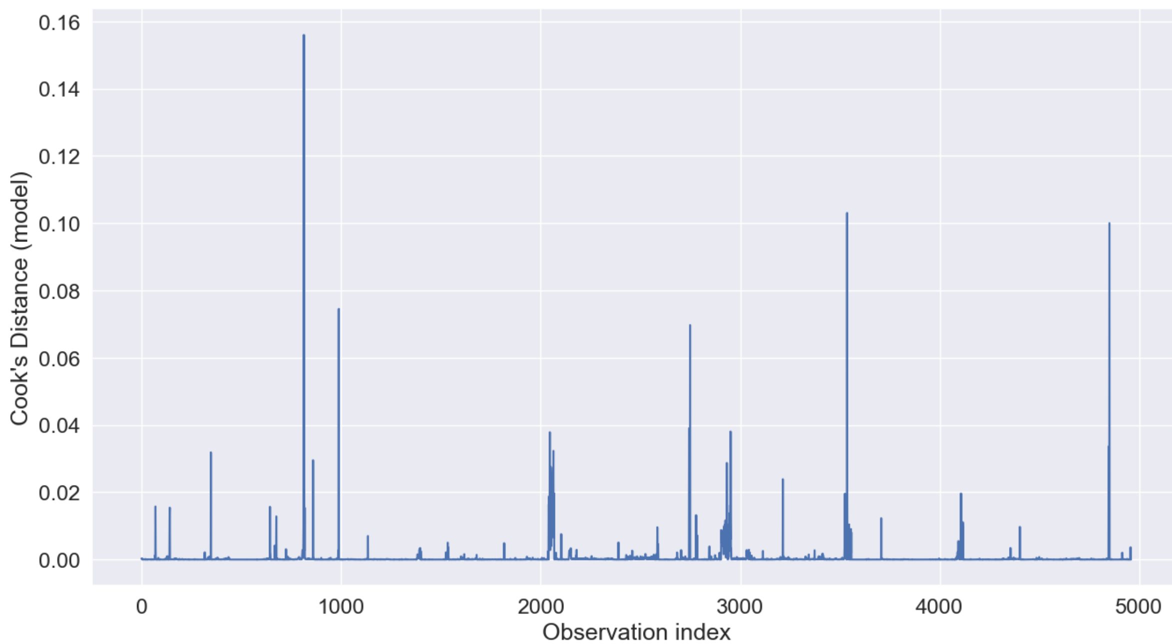
It can be shown that:

$$D_i = \frac{e_i^2}{pMSE}\left[\frac{h_i}{(1-h_i)^2}\right] \tag{6.6}$$

The larger h_i or e_i, the larger is D_i. D_i can be related to the $F(p, n-p)$ distribution. If the percentile value is 50% or more, the observation is considered as highly influential.

Cook's distance is considered high if it is greater than 0.5 and extreme if it is greater than 1.

```
sns.set(font_scale =1.5)
sns.lineplot(x = range(train.shape[0]), y = influence.cooks_distance[0])
plt.xlabel('Observation index')
plt.ylabel("Cook's Distance (model)");
```




```
# Point with the highest Cook's distance
np.where(influence.cooks_distance[0]>0.15)
```

```
(array([813], dtype=int64),)
```

The critical Cook's distance value for a point to be highly influential in this dataset is:

```
stats.f.ppf(0.5, 11, 4949)
```

```
0.9402181103263811
```

Thus, we don't have any highly influential points in the dataset.

6.3.3 Influence on regression coefficients (DFBETAS)

- $DFBETAS$ measures the influence of the i^{th} observation on the regression coefficient.
- $DFBETAS$ of the i^{th} observation on the k^{th} regression coefficient is:

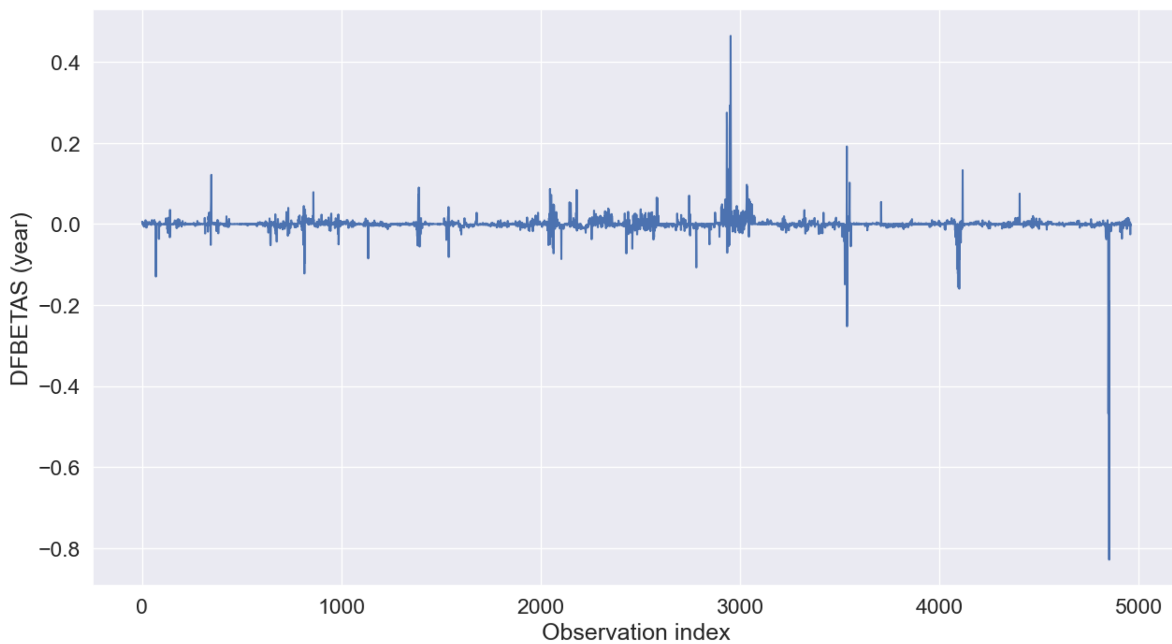
$$(DFBETAS)_{k(i)} = \frac{\hat{\beta}_k - \hat{\beta}_{k(i)}}{\sqrt{MSE_i c_k}} \tag{6.7}$$

where c_k is the k^{th} diagonal element of $(X^TX)^{-1}$.

For large datasets, an observation is considered influential if $DFBETAS$ exceeds $\frac{2}{\sqrt{n}}$.

Below is the plot of $DFBETAS$ for the `year` predictor against the observation index.

```
sns.set(font_scale =1.5)
sns.lineplot(x = range(train.shape[0]), y = influence.dfbetas[:,1])
plt.xlabel('Observation index')
plt.ylabel("DFBETAS (year)");
```



Let us analyze the point with the highest magnitude of $DFBETAS$.

```
np.where(influence.dfbetas[:,1]<-0.8)
```

```
(array([4851], dtype=int64),)
```

```
train.year.describe()
```

```
count    4960.000000
mean     2016.737903
std         2.884035
min      1997.000000
25%      2016.000000
50%      2017.000000
75%      2019.000000
max      2020.000000
Name: year, dtype: float64
```

```
train.loc[train.year<=2002,:]
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
330	13200	audi	A8	1997	Automatic	122000	Petrol	265	19.3511	4.2	465
732	13988	vw	Beetle	2001	Manual	47729	Petrol	330	32.5910	2.0	249
3157	18794	ford	Puma	2002	Manual	108000	Petrol	230	38.5757	1.7	219
3525	19395	merc	S Class	2001	Automatic	108800	Diesel	325	31.5473	3.2	169
3532	17531	merc	S Class	1999	Automatic	34000	Petrol	145	24.8735	3.2	599
3533	18761	merc	S Class	2001	Automatic	66000	Petrol	570	24.7744	3.2	449
3535	18813	merc	S Class	1998	Automatic	43534	Petrol	265	23.2962	6.0	199
3536	17891	merc	S Class	2002	Automatic	24000	Petrol	570	20.7968	5.0	699
3707	18746	hyundi	Santa Fe	2002	Manual	94000	Petrol	325	30.2671	2.4	120
4091	12995	merc	SLK	1998	Automatic	113557	Petrol	265	31.8368	2.3	199
4094	19585	merc	SLK	2001	Automatic	69234	Petrol	325	30.8839	2.0	399
4096	14265	merc	SLK	2001	Automatic	48172	Petrol	325	29.7058	2.3	399
4097	15821	merc	SLK	2002	Automatic	61400	Petrol	325	29.6568	2.3	399
4098	13021	merc	SLK	2001	Automatic	91000	Petrol	325	30.3248	2.3	395
4099	12660	merc	SLK	2001	Automatic	42087	Petrol	325	29.9404	2.3	449
4101	17521	merc	SLK	2002	Automatic	75034	Petrol	325	30.1380	2.3	499
4107	13977	merc	SLK	2000	Automatic	87000	Petrol	265	27.2998	3.2	149
4108	18679	merc	SLK	2000	Automatic	113237	Petrol	270	26.8765	3.2	399
4109	14598	merc	SLK	2001	Automatic	64476	Petrol	325	27.4628	3.2	499
4847	17268	bmw	Z3	1997	Manual	49000	Petrol	270	34.9548	1.9	395
4848	12137	bmw	Z3	1999	Manual	58000	Petrol	270	35.3077	1.9	395
4849	13288	bmw	Z3	1999	Manual	74282	Petrol	245	35.4143	1.9	399
4850	19172	bmw	Z3	2001	Manual	60000	Petrol	325	30.7305	2.2	595
4851	16577	bmw	Z3	2002	Automatic	16500	Petrol	325	29.7614	2.2	149

Let us see the leverage and studentized residual for this observation.

```
# Leverage
leverage[4851]
```

```
0.047120455781282225
```

```
# Studentized residual
out.student_resid[4851]
```

```
4.938606329343604
```

Do you see what makes this point influential?

6.4 Collinearity

Collinearity refers to the situation when two or more predictor variables have a high linear association. Linear association between a pair of variables can be measured by the correlation coefficient. Thus the correlation matrix can indicate some potential collinearity problems.

6.4.1 Why and how is collinearity a problem

(Source: page 100-101 of book)

The presence of collinearity can pose problems in the regression context, since it can be difficult to separate out the individual effects of collinear variables on the response.

Since collinearity reduces the accuracy of the estimates of the regression coefficients, it causes the standard error for $\hat{\beta}_j$ to grow. Recall that the t-statistic for each predictor is calculated by dividing $\hat{\beta}_j$ by its standard error. Consequently, collinearity results in a decline in the t-statistic. As a result, **in the presence of collinearity, we may fail to reject $H_0 : \beta_j = 0$. This means that the power of the hypothesis test—the probability of correctly detecting a non-zero coefficient—is reduced by collinearity.**

6.4.2 How to measure collinearity/multicollinearity

(Source: page 102 of book)

Unfortunately, not all collinearity problems can be detected by inspection of the correlation matrix: it is possible for collinearity to exist between three or more variables even if no pair of variables has a particularly high correlation. We call this situation multicollinearity. Instead of inspecting the correlation matrix, a better way to assess multicollinearity is to compute the variance inflation factor (VIF). The VIF is variance inflation factor the ratio of the variance of $\hat{\beta}_j$ when fitting the full model divided by the variance of $\hat{\beta}_j$ if fit on its own. The smallest possible value for VIF is 1, which indicates the complete absence of collinearity. Typically in practice there is a small amount of collinearity among the predictors. As a rule of thumb, a **VIF value that exceeds 5 or 10 indicates a problematic amount of collinearity.**

The estimated variance of the coefficient β_j, of the j^{th} predictor X_j, can be expressed as:

$$v\hat{a}r(\hat{\beta}_j) = \frac{(\hat{\sigma})^2}{(n-1)v\hat{a}r(X_j)}.\frac{1}{1-R^2_{X_j|X_{-j}}},$$

where $R^2_{X_j|X_{-j}}$ is the R-squared for the regression of X_j on the other covariates (a regression that does not involve the response variable Y).

In case of simple linear regression, the variance expression in the equation above does not contain the term $\frac{1}{1-R^2_{X_j|X_{-j}}}$, as there is only one predictor. However, in case of multiple linear regression, the variance of the estimate of the j^{th} coefficient ($\hat{\beta}_j$) gets inflated by a factor of $\frac{1}{1-R^2_{X_j|X_{-j}}}$ *(Note that in the complete absence of collinearity,* $R^2_{X_j|X_{-j}} = 0$*, and the value of this factor will be 1).*

Thus, the Variance inflation factor, or the VIF for the estimated coefficient of the j^{th} predictor X_j is:

$$VIF(\hat{\beta}_j) = \frac{1}{1-R^2_{X_j|X_{-j}}} \tag{6.8}$$

```
#Correlation matrix
train.corr()
```

	carID	year	mileage	tax	mpg	engineSize	price
carID	1.000000	0.006251	-0.001320	0.023806	-0.010774	0.011365	0.012129
year	0.006251	1.000000	-0.768058	-0.205902	-0.057093	0.014623	0.501296
mileage	-0.001320	-0.768058	1.000000	0.133744	0.125376	-0.006459	-0.478705

	carID	year	mileage	tax	mpg	engineSize	price
tax	0.023806	-0.205902	0.133744	1.000000	-0.488002	0.465282	0.144652
mpg	-0.010774	-0.057093	0.125376	-0.488002	1.000000	-0.419417	-0.369919
engineSize	0.011365	0.014623	-0.006459	0.465282	-0.419417	1.000000	0.624899
price	0.012129	0.501296	-0.478705	0.144652	-0.369919	0.624899	1.000000

Let us compute the Variance Inflation Factor (VIF) for the four predictors.

```
X = train[['mpg','year','mileage','engineSize']]
```

```
X.columns[1:]
```

```
Index(['year', 'mileage', 'engineSize'], dtype='object')
```

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tools.tools import add_constant
X = add_constant(X)
vif_data = pd.DataFrame()
vif_data["feature"] = X.columns

for i in range(len(X.columns)):
    vif_data.loc[i,'VIF'] = variance_inflation_factor(X.values, i)

print(vif_data)
```

```
      feature           VIF
0       const  1.201579e+06
1         mpg  1.243040e+00
2        year  2.452891e+00
3     mileage  2.490210e+00
4  engineSize  1.219170e+00
```

As all the values of VIF are close to one, we do not have the problem of multicollinearity in the model. Note that the VIF of `year` and `mileage` is relatively high as they are the most correlated.

Q1: Why is the VIF of the constant so high?

Q2: Why do we need to include the constant while finding the VIF?

6.4.3 Manual computation of VIF

```
#Manually computing the VIF for year
ols_object = smf.ols(formula = 'price~mpg', data = train)
model_log = ols_object.fit()
model_log.summary()
```

Dep. Variable:	price	**R-squared:**	0.137
Model:	OLS	**Adj. R-squared:**	0.137
Method:	Least Squares	**F-statistic:**	786.0
Date:	Wed, 06 Mar 2024	**Prob (F-statistic):**	1.14e-160
Time:	17:04:39	**Log-Likelihood:**	-54812.
No. Observations:	4960	**AIC:**	1.096e+05
Df Residuals:	4958	**BIC:**	1.096e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	4.144e+04	676.445	61.258	0.000	4.01e+04	4.28e+04
mpg	-374.2975	13.351	-28.036	0.000	-400.471	-348.124

Omnibus:	2132.208	**Durbin-Watson:**	0.320
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	13751.995
Skew:	1.942	**Prob(JB):**	0.00
Kurtosis:	10.174	**Cond. No.**	158.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

```
(13.351/9.338)**2
```

```
2.044183378279958
```

```
#Manually computing the VIF for year
ols_object = smf.ols(formula = 'price~year+mpg+engineSize+mileage', data = train)
model_log = ols_object.fit()
model_log.summary()
```

Dep. Variable:	price	**R-squared:**	0.660
Model:	OLS	**Adj. R-squared:**	0.660
Method:	Least Squares	**F-statistic:**	2410.
Date:	Wed, 06 Mar 2024	**Prob (F-statistic):**	0.00
Time:	17:01:18	**Log-Likelihood:**	-52497.
No. Observations:	4960	**AIC:**	1.050e+05
Df Residuals:	4955	**BIC:**	1.050e+05
Df Model:	4		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	-3.661e+06	1.49e+05	-24.593	0.000	-3.95e+06	-3.37e+06
year	1817.7366	73.751	24.647	0.000	1673.151	1962.322
mpg	-79.3126	9.338	-8.493	0.000	-97.620	-61.006
engineSize	1.218e+04	189.969	64.107	0.000	1.18e+04	1.26e+04
mileage	-0.1474	0.009	-16.817	0.000	-0.165	-0.130

Omnibus:	2450.973	**Durbin-Watson:**	0.541
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	31060.548
Skew:	2.045	**Prob(JB):**	0.00
Kurtosis:	14.557	**Cond. No.**	3.83e+07

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.83e+07. This might indicate that there are strong multicollinearity or other numerical problems.

```
#Manually computing the VIF for year
ols_object = smf.ols(formula = 'year~mpg+engineSize+mileage', data = train)
model_log = ols_object.fit()
model_log.summary()
```

Dep. Variable:	year	**R-squared:**	0.592
Model:	OLS	**Adj. R-squared:**	0.592
Method:	Least Squares	**F-statistic:**	2400.
Date:	Wed, 06 Mar 2024	**Prob (F-statistic):**	0.00
Time:	17:00:13	**Log-Likelihood:**	-10066.
No. Observations:	4960	**AIC:**	2.014e+04
Df Residuals:	4956	**BIC:**	2.017e+04
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P> |t|	[0.025	0.975]
Intercept	2018.3135	0.140	1.44e+04	0.000	2018.039	2018.588
mpg	0.0095	0.002	5.301	0.000	0.006	0.013
engineSize	0.1171	0.037	3.203	0.001	0.045	0.189
mileage	-9.139e-05	1.08e-06	-84.615	0.000	-9.35e-05	-8.93e-05

Omnibus:	2949.664	**Durbin-Watson:**	1.161
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	63773.271
Skew:	-2.426	**Prob(JB):**	0.00
Kurtosis:	19.883	**Cond. No.**	1.91e+05

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.91e+05. This might indicate that there are
strong multicollinearity or other numerical problems.

```
#VIF for year
1/(1-0.592)
```

```
2.4509803921568625
```

Note that year and mileage have a high linear correlation. Removing one of them should decrease the standard error of the coefficient of the other, without significanty decrease R-squared.

```
ols_object = smf.ols(formula = 'price~mpg+engineSize+mileage+year', data = train)
model_log = ols_object.fit()
model_log.summary()
```

Table 6.6: OLS Regression Results

Dep. Variable:	price	R-squared:	0.660
Model:	OLS	Adj. R-squared:	0.660
Method:	Least Squares	F-statistic:	2410.
Date:	Tue, 07 Feb 2023	Prob (F-statistic):	0.00
Time:	21:39:45	Log-Likelihood:	-52497.
No. Observations:	4960	AIC:	1.050e+05
Df Residuals:	4955	BIC:	1.050e+05
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-3.661e+06	1.49e+05	-24.593	0.000	-3.95e+06	-3.37e+06
mpg	-79.3126	9.338	-8.493	0.000	-97.620	-61.006
engineSize	1.218e+04	189.969	64.107	0.000	1.18e+04	1.26e+04
mileage	-0.1474	0.009	-16.817	0.000	-0.165	-0.130
year	1817.7366	73.751	24.647	0.000	1673.151	1962.322

Omnibus:	2450.973	Durbin-Watson:	0.541
Prob(Omnibus):	0.000	Jarque-Bera (JB):	31060.548
Skew:	2.045	Prob(JB):	0.00
Kurtosis:	14.557	Cond. No.	3.83e+07

Removing mileage from the above regression.

```
ols_object = smf.ols(formula = 'price~mpg+engineSize+year', data = train)
model_log = ols_object.fit()
model_log.summary()
```

Table 6.9: OLS Regression Results

Dep. Variable:	price	R-squared:	0.641
Model:	OLS	Adj. R-squared:	0.641
Method:	Least Squares	F-statistic:	2951.
Date:	Tue, 07 Feb 2023	Prob (F-statistic):	0.00
Time:	21:40:00	Log-Likelihood:	-52635.
No. Observations:	4960	AIC:	1.053e+05
Df Residuals:	4956	BIC:	1.053e+05
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-5.586e+06	9.78e+04	-57.098	0.000	-5.78e+06	-5.39e+06
mpg	-101.9120	9.500	-10.727	0.000	-120.536	-83.288
engineSize	1.196e+04	194.848	61.392	0.000	1.16e+04	1.23e+04
year	2771.1844	48.492	57.147	0.000	2676.118	2866.251

Omnibus:	2389.075	Durbin-Watson:	0.528
Prob(Omnibus):	0.000	Jarque-Bera (JB):	26920.051
Skew:	2.018	Prob(JB):	0.00
Kurtosis:	13.675	Cond. No.	1.41e+06

Note that the standard error of the coefficient of *year* has reduced from 73 to 48, without any large reduction in R-squared.

6.4.4 When can we overlook multicollinearity?

- The severity of the problems increases with the degree of the multicollinearity. Therefore, if there is only moderate multicollinearity *(5 < VIF < 10)*, we may overlook it.
- Multicollinearity affects only the standard errors of the coefficients of collinear predictors. Therefore, if multicollinearity is not present for the predictors that we are particularly interested in, we may not need to resolve it.
- Multicollinearity affects the standard error of the coefficients and thereby their p-values, but in general, it does not influence the prediction accuracy, except in the case that the coefficients are so unstable that the predictions are outside of the domain space of the response. If our sole aim is prediction, and we don't wish to infer the statistical significance of predictors, then we may avoid addressing multicollinearity. "*The fact that some or all predictor variables are correlated among themselves does not, in general, inhibit our ability to obtain a good fit nor does it tend to affect inferences about mean responses or predictions of new observations, provided these inferences are made within the region of observations" - Neter, John, Michael H. Kutner, Christopher J. Nachtsheim, and William Wasserman. "Applied linear statistical models." (1996): 318.*

7 Logistic regression: Introduction and Metrics

Read sections 4.1 - 4.3 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

7.1 Theory Behind Logistic Regression

Logistic regression is the go-to linear classification algorithm for two-class problems. It is easy to implement, easy to understand and gets great results on a wide variety of problems, even when the expectations the method has for your data are violated.

7.1.1 Description

Logistic regression is named for the function used at the core of the method, the logistic function.

The logistic function, also called the **`Sigmoid function`** was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It's an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

$$\frac{1}{1+e^{-x}}$$

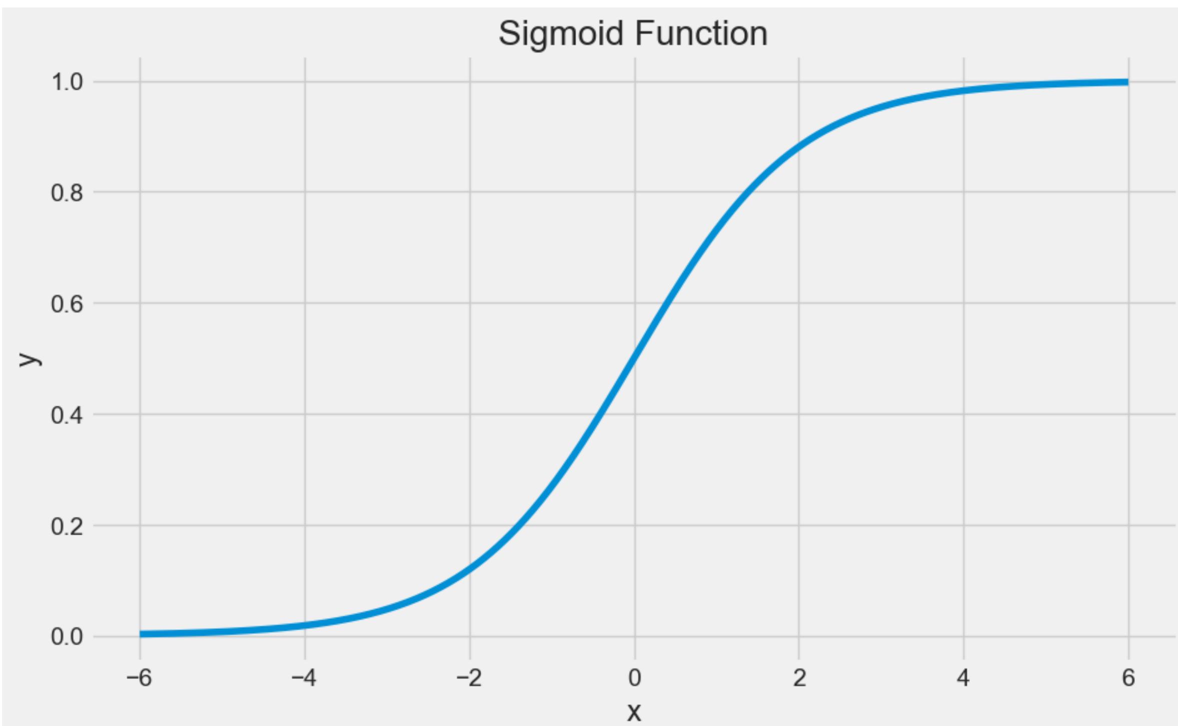
e is the base of the natural logarithms and x is value that you want to transform via the logistic function.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.formula.api as sm
```

```
from sklearn.metrics import precision_recall_curve, roc_curve, auc, accuracy_score
from sklearn.linear_model import LogisticRegression
```

```
%matplotlib inline
sns.set_style('whitegrid')
plt.style.use("fivethirtyeight")
x = np.linspace(-6, 6, num=1000)
plt.figure(figsize=(10, 6))
plt.plot(x, (1 / (1 + np.exp(-x))))
plt.xlabel("x")
plt.ylabel("y")
plt.title("Sigmoid Function")
```

```
Text(0.5, 1.0, 'Sigmoid Function')
```



The logistic regression equation has a very similar representation like linear regression. The difference is that the output value being modelled is binary in nature.

$$\hat{p} = \frac{e^{\hat{\beta}_0 + \hat{\beta}_1 x_1}}{1 + e^{\hat{\beta}_0 + \hat{\beta}_1 x_1}}$$

or

$$\hat{p} = \frac{1.0}{1.0 + e^{-(\hat{\beta}_0 + \hat{\beta}_1 x_1)}}$$

$\hat{\beta}_0$ is the estimated intercept term

$\hat{\beta}_1$ is the estimated coefficient for x_1

$\hat{p}$ is the predicted output with real value between 0 and 1. To convert this to binary output of 0 or 1, this would either need to be rounded to an integer value or a cutoff point be provided to specify the class segregation point.

7.1.2 Learning the Logistic Regression Model

The coefficients (Beta values b) of the logistic regression algorithm must be estimated from your training data. This is done using maximum-likelihood estimation.

Maximum-likelihood estimation is a common learning algorithm used by a variety of machine learning algorithms, although it does make assumptions about the distribution of your data (more on this when we talk about preparing your data).

The best coefficients should result in a model that would predict a value very close to 1 (e.g. male) for the default class and a value very close to 0 (e.g. female) for the other class. The intuition for maximum-likelihood for logistic regression is that a search procedure seeks values for the coefficients (Beta values) that maximize the likelihood of the observed data. In other words, in MLE, we estimate the parameter values (Beta values) which are the most likely to produce that data at hand.

Here is an analogy to understand the idea behind Maximum Likelihood Estimation (MLE). Let us say, you are listening to a song (data). You are not aware of the singer (parameter) of the song. With just the musical piece at hand, you try to guess the singer (parameter) who you feel is the most likely (MLE) to have sung that song. Your are making a maximum likelihood estimate! Out of all the singers (parameter space) you have chosen them as the one who is the most likely to have sung that song (data).

We are not going to go into the math of maximum likelihood. It is enough to say that a minimization algorithm is used to optimize the best values for the coefficients for your training data. This is often implemented in practice using efficient numerical optimization algorithm (like the Quasi-newton method).

When you are learning logistic, you can implement it yourself from scratch using the much simpler gradient descent algorithm.

7.1.3 Preparing Data for Logistic Regression

The assumptions made by logistic regression about the distribution and relationships in your data are much the same as the assumptions made in linear regression.

Much study has gone into defining these assumptions and precise probabilistic and statistical language is used. My advice is to use these as guidelines or rules of thumb and experiment with different data preparation schemes.

Ultimately in predictive modeling machine learning projects you are laser focused on making accurate predictions rather than interpreting the results. As such, you can break some assumptions as long as the model is robust and performs well.

- **Binary Output Variable:** This might be obvious as we have already mentioned it, but logistic regression is intended for binary (two-class) classification problems. It will predict the probability of an instance belonging to the default class, which can be snapped into a 0 or 1 classification.
- **Remove Noise:** Logistic regression assumes no error in the output variable (y), consider removing outliers and possibly misclassified instances from your training data.
- **Gaussian Distribution:** Logistic regression is a linear algorithm (with a non-linear transform on output). It does assume a linear relationship between the input variables with the output. Data transforms of your input variables that better expose this linear relationship can result in a more accurate model. For example, you can use log, root, Box-Cox and other univariate transforms to better expose this relationship.
- **Remove Correlated Inputs:** Like linear regression, the model can overfit if you have multiple highly-correlated inputs. Consider calculating the pairwise correlations between all inputs and removing highly correlated inputs.
- **Fail to Converge:** It is possible for the expected likelihood estimation process that learns the coefficients to fail to converge. This can happen if there are many highly correlated inputs in your data or the data is very sparse (e.g. lots of zeros in your input data).

7.2 Logistic Regression: Scikit-learn vs Statsmodels

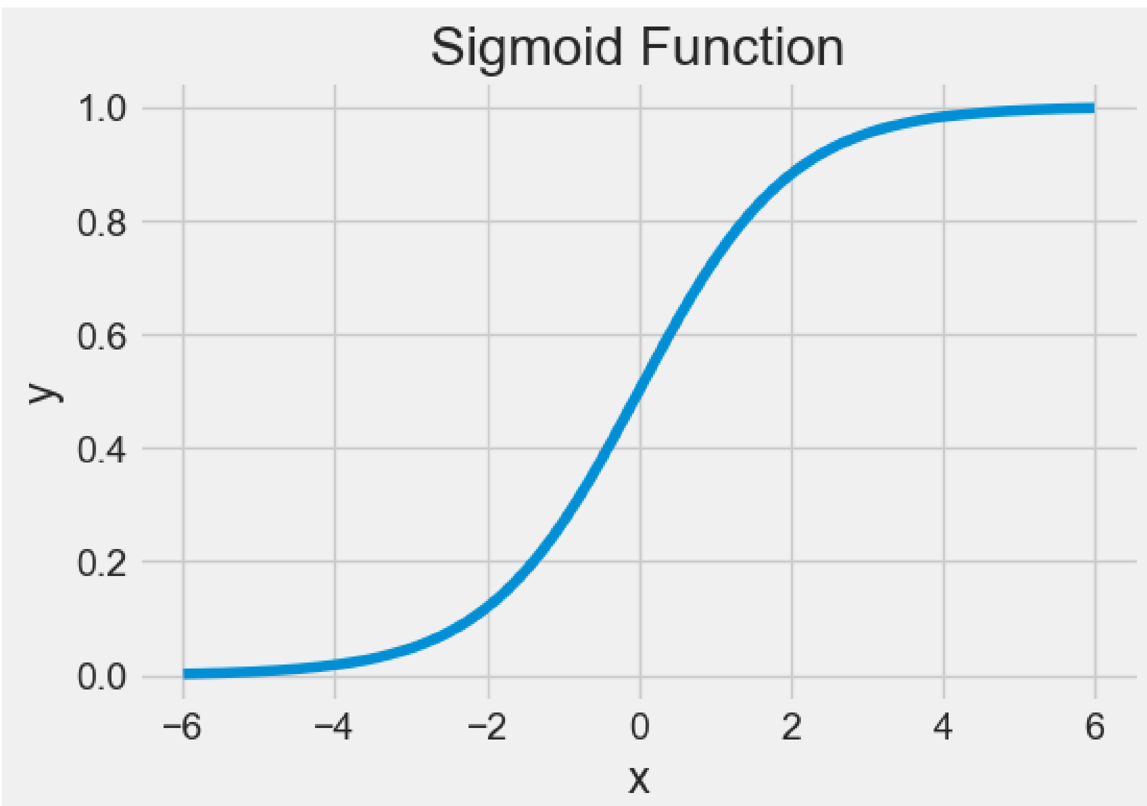
Python gives us two ways to do logistic regression. Statsmodels offers modeling from the perspective of statistics. Scikit-learn offers some of the same models from the perspective of machine learning.

So we need to understand the difference between statistics and machine learning! Statistics makes mathematically valid inferences about a population based on sample data. Statistics answers the question, “What is the evidence that X is related to Y?” Machine learning has the goal of optimizing predictive accuracy rather than inference. Machine learning answers the question, “Given X, what prediction should we make for Y?”

7.3 Training a logistic regression model

Read the data on social network ads. The data shows if the person purchased a product when targeted with an ad on social media. Fit a logistic regression model to predict if a user will purchase the product based on their characteristics such as age, gender and estimated salary.

```
%matplotlib inline
sns.set_style('whitegrid')
plt.style.use("fivethirtyeight")
x = np.linspace(-6, 6, num=1000)
plt.figure(figsize=(6, 4))
plt.plot(x, (1 / (1 + np.exp(-x))))
plt.xlabel("x")
plt.ylabel("y")
plt.title("Sigmoid Function");
```



7.4 Logistic Regression: Scikit-learn vs Statsmodels

Python gives us two ways to do logistic regression. Statsmodels offers modeling from the perspective of statistics. Scikit-learn offers some of the same models from the perspective of machine learning.

So we need to understand the difference between statistics and machine learning! Statistics makes mathematically valid inferences about a population based on sample data. Statistics answers the question, "What is the evidence that X is related to Y?" Machine learning has the goal of optimizing predictive accuracy rather than inference. Machine learning answers the question, "Given X, what prediction should we make for Y?"

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.formula.api as sm
from sklearn.metrics import precision_recall_curve, roc_curve, auc, accuracy_score
from sklearn.linear_model import LogisticRegression
```

Read the data on social network ads. The data shows if the person purchased a product when targeted with an ad on social media. Fit a logistic regression model to predict if a user will purchase the product based on their characteristics such as age, gender and estimated salary.

```
train = pd.read_csv('./Datasets/Social_Network_Ads_train.csv') #Develop the model on train da
test = pd.read_csv('./Datasets/Social_Network_Ads_test.csv') #Test the model on test data
```

```
train.head()
```

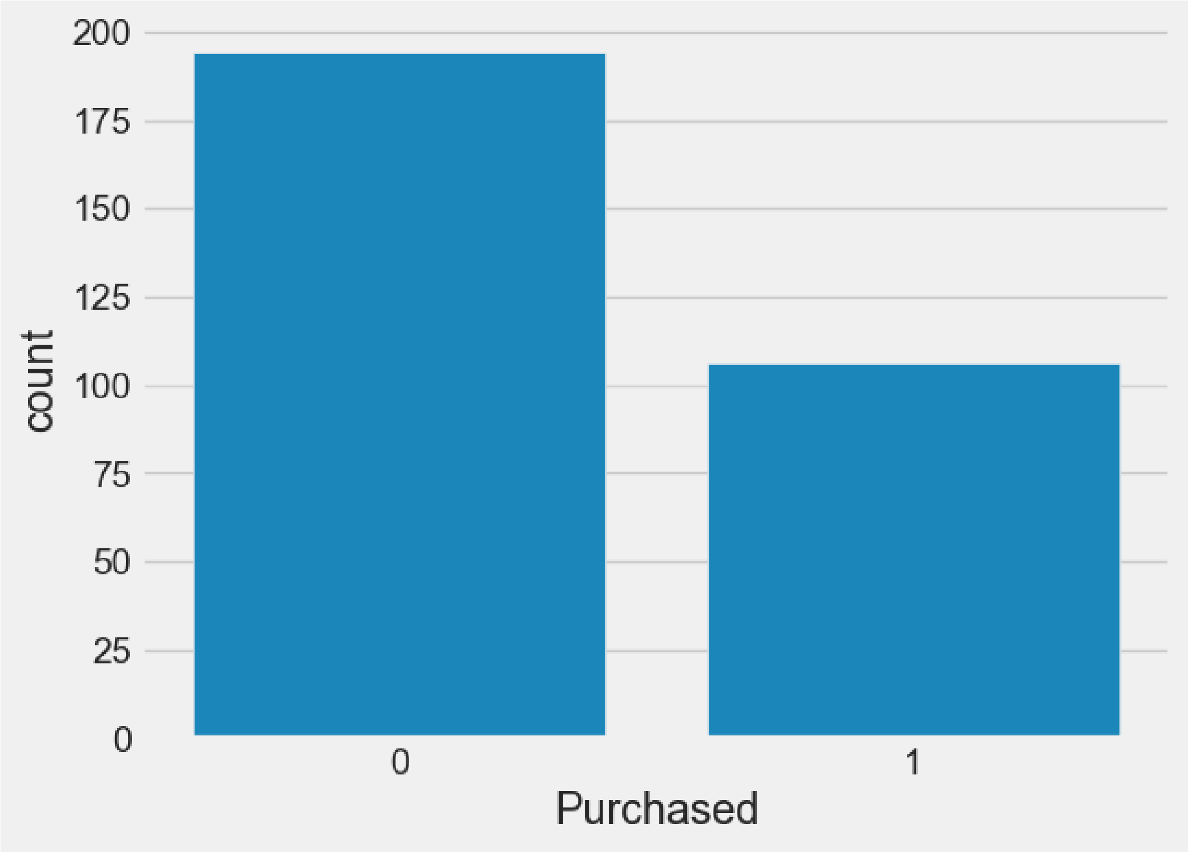
	User ID	Gender	Age	EstimatedSalary	Purchased
0	15755018	Male	36	33000	0
1	15697020	Female	39	61000	0
2	15796351	Male	36	118000	1
3	15665760	Male	39	122000	1
4	15794661	Female	26	118000	0

7.4.1 Examining the Distribution of the Target Column, make sure our target is not severely imbalanced

```
train.Purchased.value_counts()
```

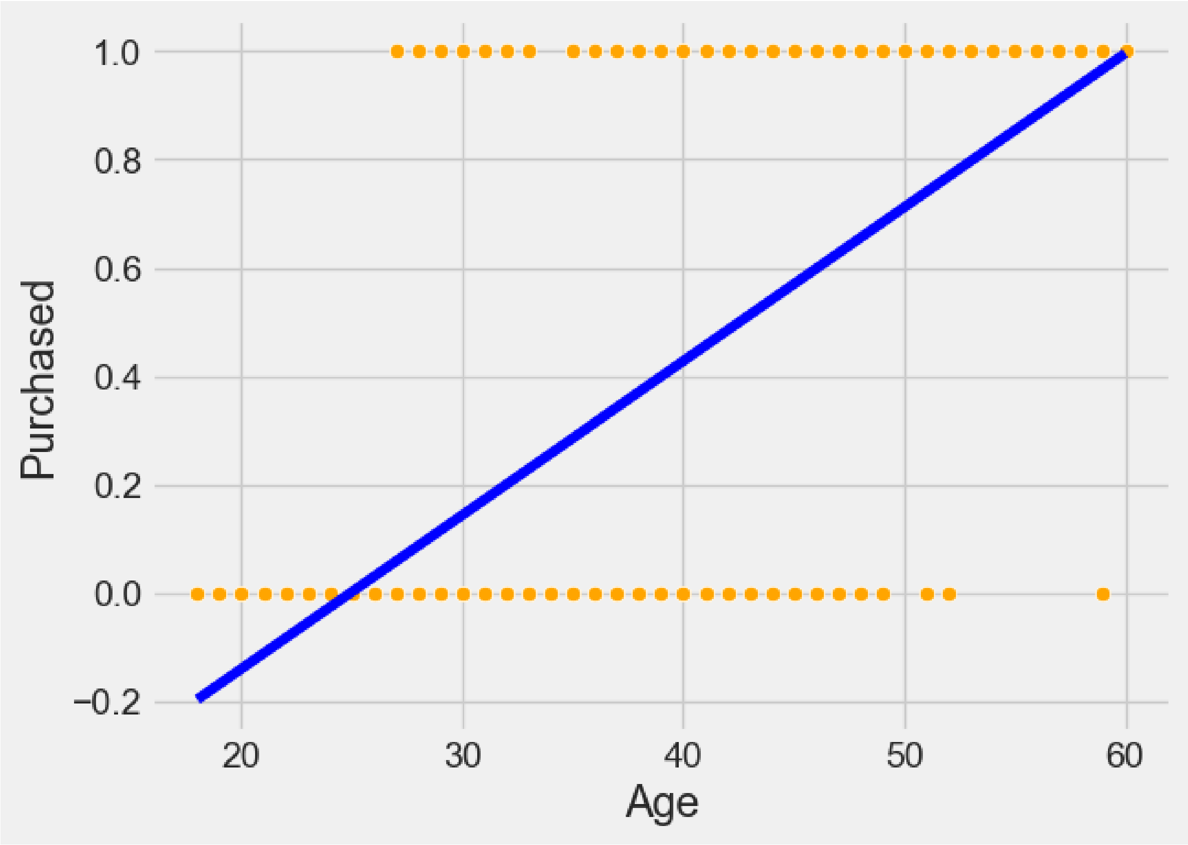
```
Purchased
0    194
1    106
Name: count, dtype: int64
```

```
sns.countplot(x = 'Purchased',data = train);
```



7.4.2 Fitting a linear regression

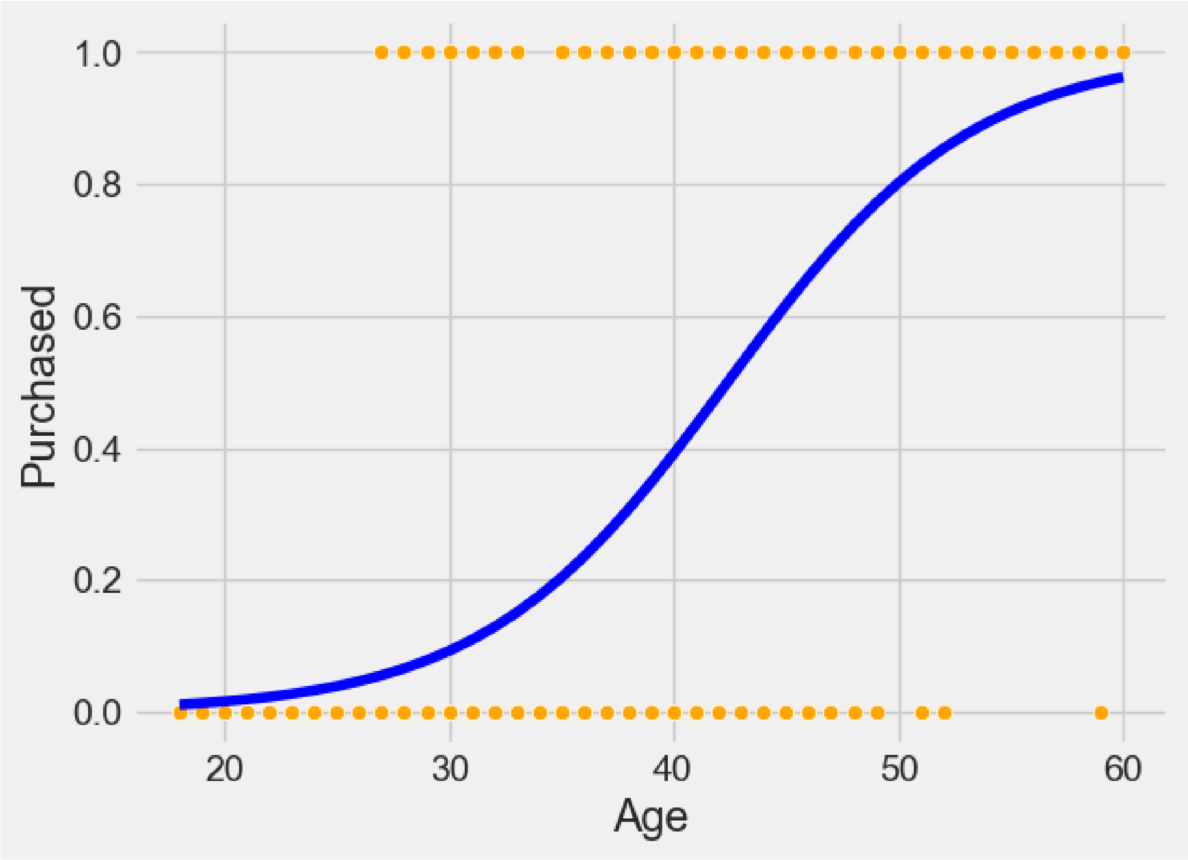
```
sns.scatterplot(x = 'Age', y = 'Purchased', data = train, color = 'orange') #Visualizing data
lm = sm.ols(formula = 'Purchased~Age', data = train).fit() #Developing linear regression mode
sns.lineplot(x = 'Age', y= lm.predict(train), data = train, color = 'blue') #Visualizing mode
```



7.4.3 Logistic Regression with Statsmodel

```
sns.scatterplot(x = 'Age', y = 'Purchased', data = train, color = 'orange') #Visualizing data
logit_model = sm.logit(formula = 'Purchased~Age', data = train).fit() #Developing logistic re
sns.lineplot(x = 'Age', y= logit_model.predict(train), data = train, color = 'blue') #Visuali
```

```
Optimization terminated successfully.
         Current function value: 0.430107
         Iterations 7
```




```
logit_model.summary()
```

Dep. Variable:	Purchased	**No. Observations:**	300
Model:	Logit	**Df Residuals:**	298
Method:	MLE	**Df Model:**	1
Date:	Sun, 09 Feb 2025	**Pseudo R-squ.:**	0.3378
Time:	18:28:20	**Log-Likelihood:**	-129.03
converged:	True	**LL-Null:**	-194.85
Covariance Type:	nonrobust	**LLR p-value:**	1.805e-30

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
Intercept	-7.8102	0.885	-8.825	0.000	-9.545	-6.076
Age	0.1842	0.022	8.449	0.000	0.141	0.227

```
logit_model_gender = sm.logit(formula = 'Purchased~Gender', data = train).fit()
logit_model_gender.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.648804
         Iterations 4
```

Dep. Variable:	Purchased	**No. Observations:**	300
Model:	Logit	**Df Residuals:**	298
Method:	MLE	**Df Model:**	1
Date:	Sun, 09 Feb 2025	**Pseudo R-squ.:**	0.001049
Time:	18:28:20	**Log-Likelihood:**	-194.64
converged:	True	**LL-Null:**	-194.85
Covariance Type:	nonrobust	**LLR p-value:**	0.5225

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
Intercept	-0.5285	0.168	-3.137	0.002	-0.859	-0.198
Gender[T.Male]	-0.1546	0.242	-0.639	0.523	-0.629	0.319

```
# Predicted probabilities
predicted_probabilities = logit_model.predict(train)
predicted_probabilities
```

```
0      0.235159
1      0.348227
2      0.235159
3      0.348227
4      0.046473
         ...
295    0.737081
296    0.481439
297    0.065810
298    0.829688
299    0.150336
Length: 300, dtype: float64
```

```
# Predicted classes (binary outcome, 0 or 1)
predicted_classes = (predicted_probabilities > 0.5).astype(int)
predicted_classes
```

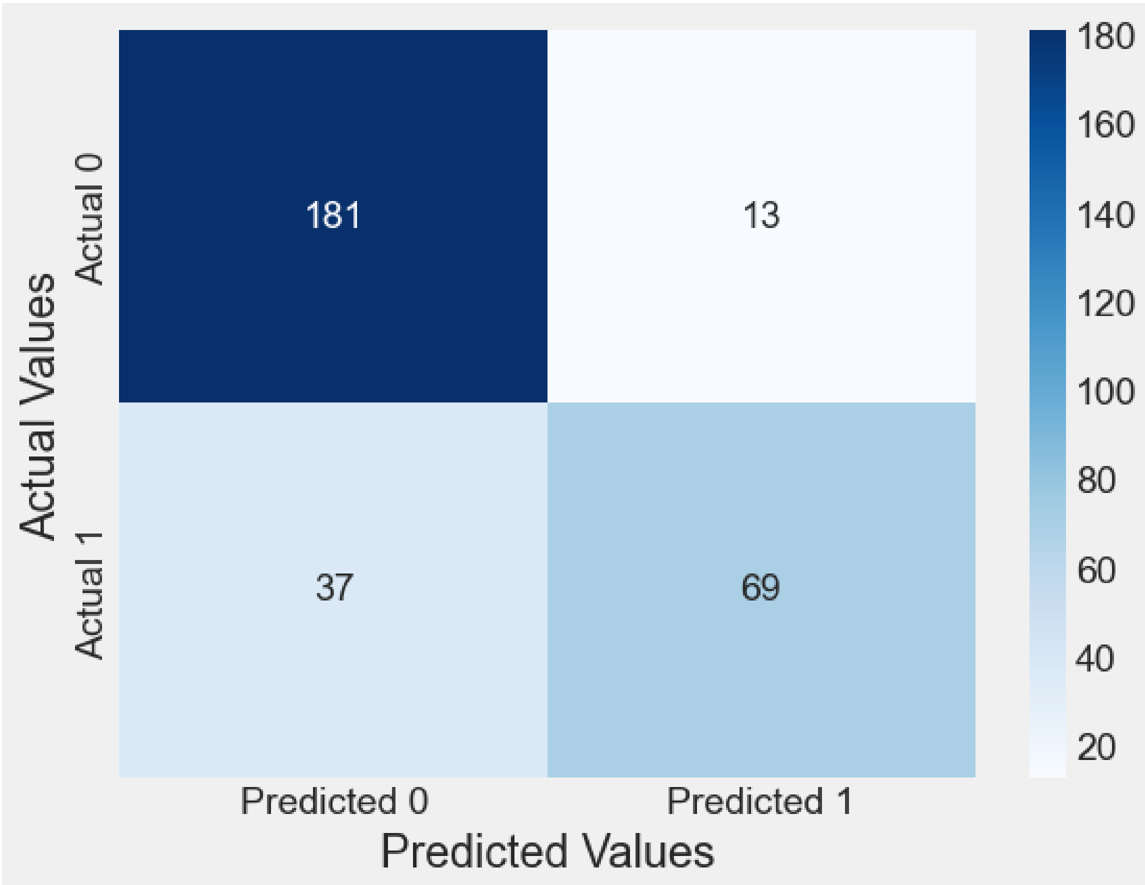
```
0      0
1      0
2      0
3      0
```

```
4      0
      ..
295    1
296    0
297    0
298    1
299    0
Length: 300, dtype: int32
```

```
#Function to compute confusion matrix and prediction accuracy on training data
def confusion_matrix_train(model,cutoff=0.5):
    # Confusion matrix
    cm_df = pd.DataFrame(model.pred_table(threshold = cutoff))
    #Formatting the confusion matrix
    cm_df.columns = ['Predicted 0', 'Predicted 1']
    cm_df = cm_df.rename(index={0: 'Actual 0',1: 'Actual 1'})
    cm = np.array(cm_df)
    # Calculate the accuracy
    accuracy = (cm[0,0]+cm[1,1])/cm.sum()
    sns.heatmap(cm_df, annot=True, cmap='Blues', fmt='g')
    plt.ylabel("Actual Values")
    plt.xlabel("Predicted Values")
    print("Classification accuracy = {:.1%}".format(accuracy))
```

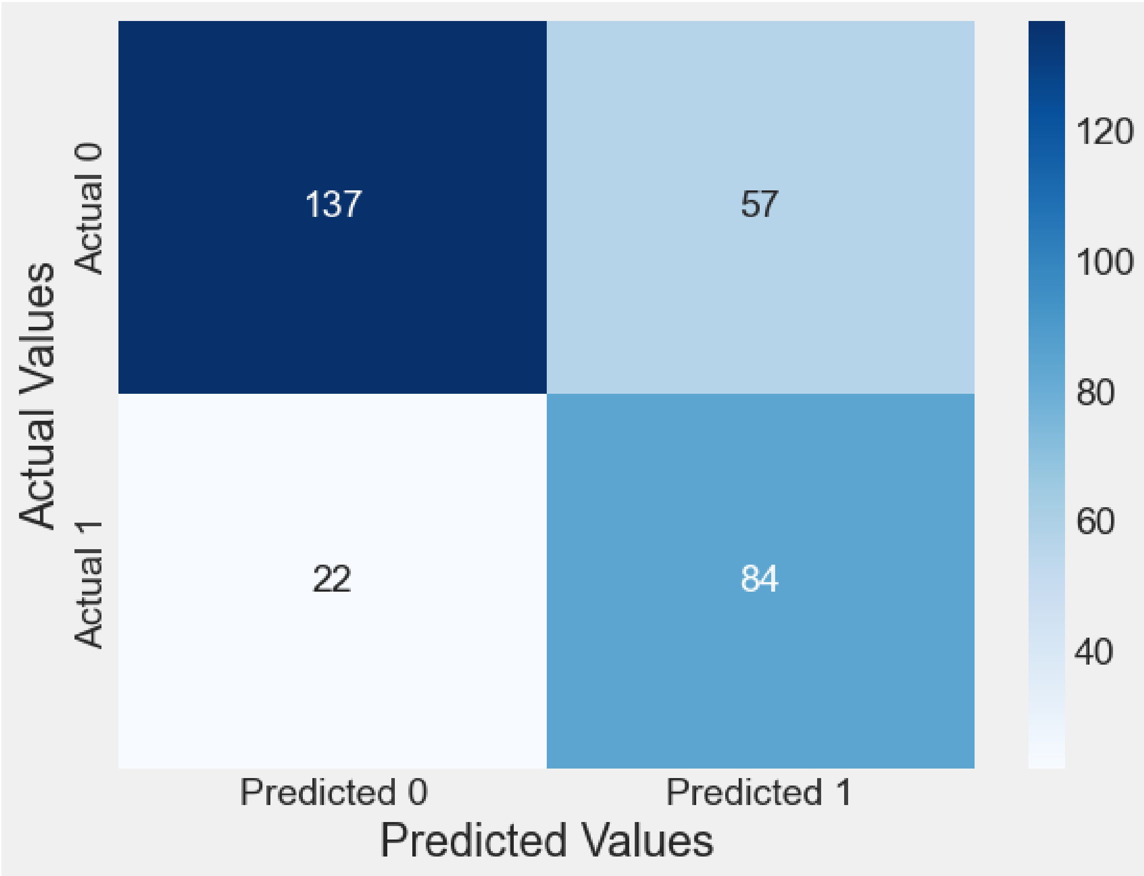
```
cm = confusion_matrix_train(logit_model)
```

```
Classification accuracy = 83.3%
```



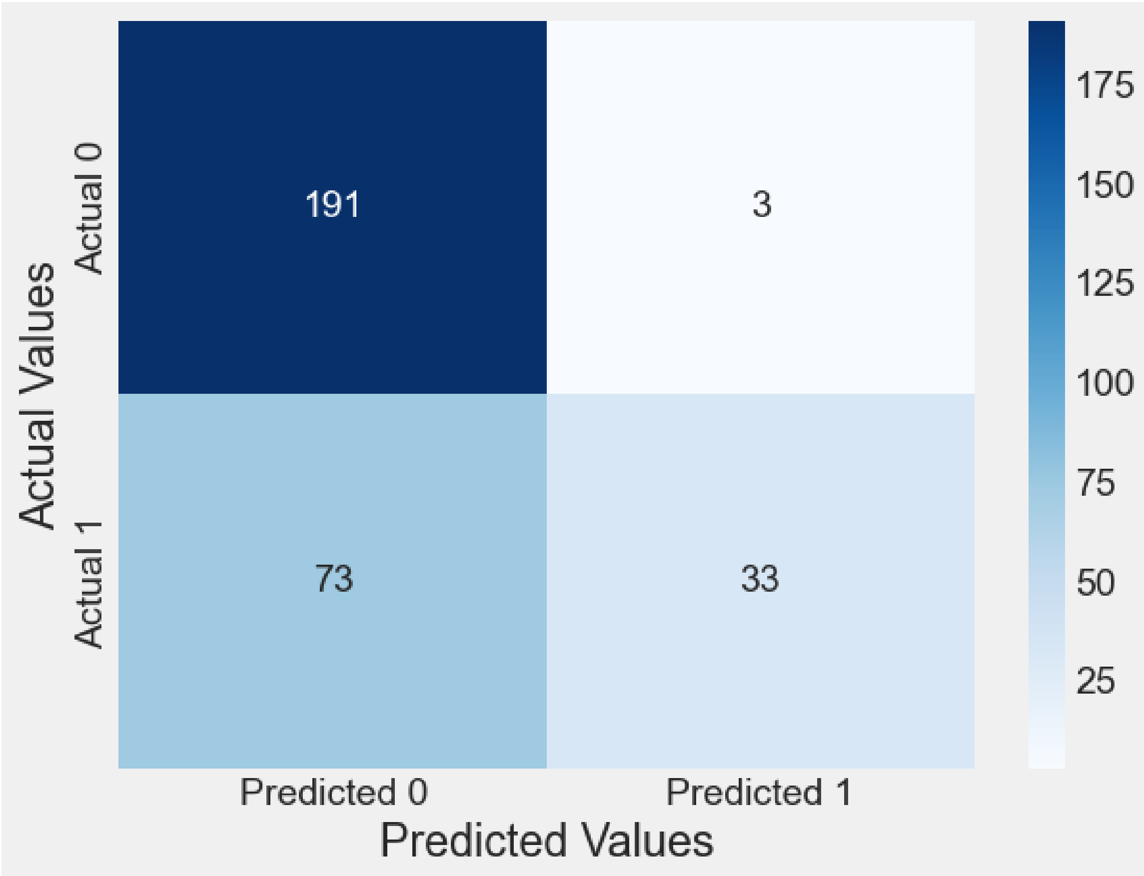

```
# change the cutoff to 0.3
cm = confusion_matrix_train(logit_model, 0.3)
```

```
Classification accuracy = 73.7%
```




```
# increase the cutoff to 0.7
cm = confusion_matrix_train(logit_model, 0.8)
```

```
Classification accuracy = 74.7%
```



Making prediction on test set and output the model's performance

```
# Predicted probabilities
predicted_probabilities = logit_model.predict(test)
```

```
# Predicted classes (binary outcome, 0 or 1)
predicted_classes = (predicted_probabilities > 0.5).astype(int)
predicted_classes
```

```
0      0
1      0
2      0
3      0
4      0
      ..
95     1
96     1
```

```
97    1
98    0
99    1
Length: 100, dtype: int32
```

```
from sklearn.metrics import confusion_matrix
```

```
confusion_mat = confusion_matrix(test.Purchased, predicted_classes)
# Define labels for the confusion matrix
labels = ['Actual Negative', 'Actual Positive']
# Create a formatted confusion matrix
formatted_confusion_mat = pd.DataFrame(confusion_mat, index=labels, columns=[f'Predicted {lab

print("Confusion Matrix:")
print(formatted_confusion_mat)
```

```
Confusion Matrix:
                 Predicted Actual Negative  Predicted Actual Positive
Actual Negative                         58                          5
Actual Positive                          9                         28
```

7.4.4 Logistic Regression with Sklearn

```
X_train = train[['Age']]
y_train = train['Purchased']

X_test = test[['Age']]
y_test = test['Purchased']
```

```
# turn off regularization
skn_model = LogisticRegression(penalty=None)
```

```
skn_model.fit(X_train, y_train)
```

```
LogisticRegression(penalty=None)
```

```
# Note that in sklearn, .predict returns the classes directly, with 0.5 threshold
y_pred_test = skn_model.predict(X_test)
y_pred_test
```

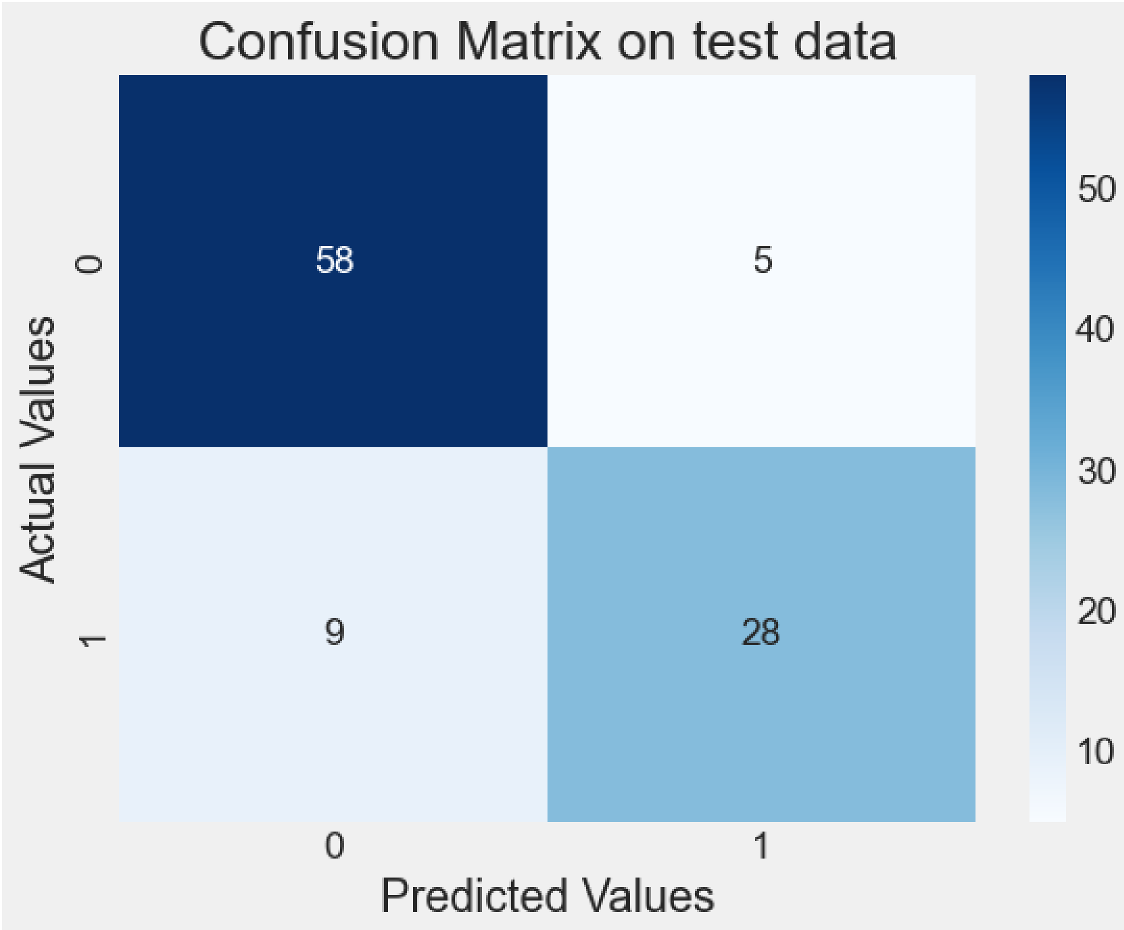
```
array([0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0,
       1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1], dtype=int64)
```

```
# To return the prediction probabilities, we need .predict_proba
# # probs_y is a 2-D array of probability of being labeled as 0 (first column of array) vs 1
y_pred_probs = skn_model.predict_proba(X_test)
y_pred_probs[:5]
```

```
array([[0.79634123, 0.20365877],
       [0.95352574, 0.04647426],
       [0.944647  , 0.055353  ],
       [0.8717078 , 0.1282922 ],
       [0.92191865, 0.07808135]])
```

```
cm=confusion_matrix(y_test,y_pred_test)
#plt.figure(figsize=(4,4))
plt.title("Confusion Matrix on test data")
sns.heatmap(cm, annot=True,fmt='d', cmap='Blues')
plt.ylabel("Actual Values")
plt.xlabel("Predicted Values")
```

```
Text(0.5, 5.183333333333314, 'Predicted Values')
```




```
from sklearn.metrics import accuracy_score
print("Accuracy:", accuracy_score(y_test, y_pred_test))
from sklearn.metrics import precision_score
print("Precision:", precision_score(y_test, y_pred_test))
from sklearn.metrics import recall_score
print("Recall:", recall_score(y_test, y_pred_test))
from sklearn.metrics import f1_score
print("F1 score:",  f1_score(y_test, y_pred_test))
```

```
Accuracy: 0.86
Precision: 0.8484848484848485
Recall: 0.7567567567567568
F1 score: 0.8
```

7.4.5 Changing the default threshold

```
new_threshold = 0.3
```

```
predicted_classes_new_threshold = (y_pred_probs > new_threshold).astype(int)
predicted_classes_new_threshold[:5]
```

```
array([[1, 0],
       [1, 0],
       [1, 0],
       [1, 0],
       [1, 0]])
```

```
confusion_mat_new_threshold = confusion_matrix(y_test, predicted_classes_new_threshold[:, 1])
print("Confusion Matrix (Threshold =", new_threshold, "):")
print(confusion_mat_new_threshold)
from sklearn.metrics import accuracy_score
print("Accuracy:", accuracy_score(y_test, predicted_classes_new_threshold[:, 1]))
from sklearn.metrics import precision_score
print("Precision:", precision_score(y_test, predicted_classes_new_threshold[:, 1]))
from sklearn.metrics import recall_score
print("Recall:", recall_score(y_test, predicted_classes_new_threshold[:, 1]))
from sklearn.metrics import f1_score
print("F1 score:",  f1_score(y_test, predicted_classes_new_threshold[:, 1]))
```

```
Confusion Matrix (Threshold = 0.3 ):
[[44 19]
 [ 7 30]]
Accuracy: 0.74
Precision: 0.6122448979591837
Recall: 0.8108108108108109
F1 score: 0.6976744186046512
```

7.5 Performance Measurement

We have already seen the confusion matrix, and classification accuracy. Now, let us see some other useful performance metrics that can be computed from the confusion matrix. The metrics below are computed for the confusion matrix immediately above this section *(or the confusion matrix on test data corresponding to the model `logit_model_diabetes`).*

7.5.1 Precision-recall

Precision measures the accuracy of positive predictions. Also called the `precision` of the classifier

$$\text{precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

`==> 70.13%`

`Precision` is typically used with `recall` (`Sensitivity` or `True Positive Rate`). The ratio of positive instances that are correctly detected by the classifier.

$\text{recall} = \frac{\text{True Positives}}{\text{True Positives+False Negatives}}$ `==> 88.52%`

Precision / Recall Tradeoff: Increasing precision reduces recall and vice versa.

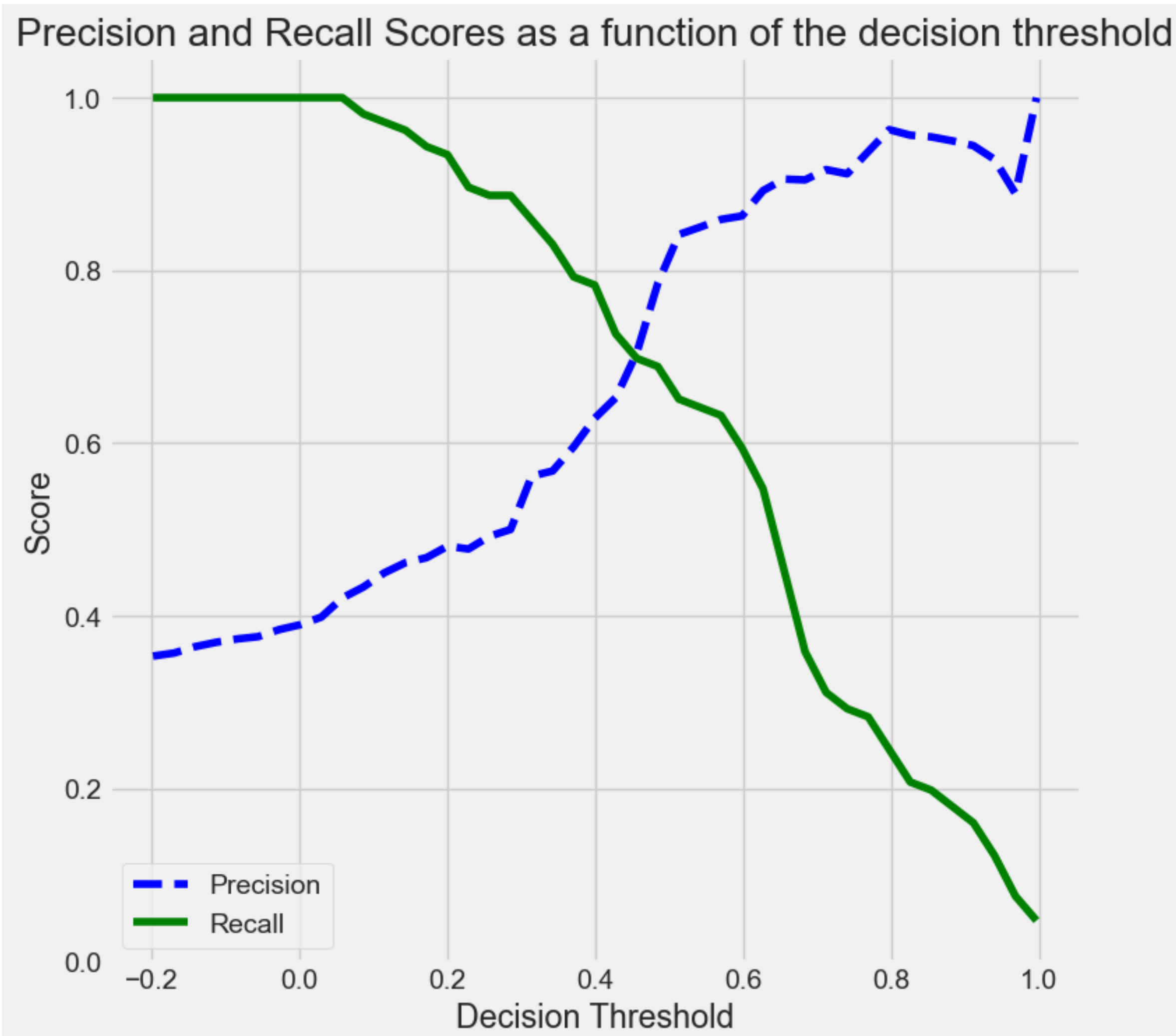
Visualize the precision-recall curve for the model `logit_model_diabetes`.

```
train
```

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15755018	Male	36	33000	0
1	15697020	Female	39	61000	0
2	15796351	Male	36	118000	1
3	15665760	Male	39	122000	1
4	15794661	Female	26	118000	0
...	...	...	...	...	...
295	15724536	Female	48	96000	1
296	15701537	Male	42	149000	1
297	15807481	Male	28	79000	0
298	15603942	Female	51	134000	0
299	15690188	Female	33	28000	0

```
y=train.Purchased
ypred = lm.predict(train)
p, r, thresholds = precision_recall_curve(y, ypred)
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.ylabel("Score")
```

```
    plt.xlabel("Decision Threshold")
    plt.legend(loc='best')
    plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)
```



As the decision threshold probability increases, the precision increases, while the recall decreases.

Q: How are the values of the `thresholds` chosen to make the precision-recall curve?

Hint: Look at the documentation for `precision_recall_curve`.

7.5.2 The Receiver Operating Characteristics (ROC) Curve

A **ROC(Receiver Operator Characteristic Curve)** is a plot of sensitivity (True Positive Rate) on the y axis against (1−specificity) (False Positive Rate) on the x axis for varying values of the threshold t. The 45° diagonal line connecting (0,0) to (1,1) is the ROC curve corresponding to random chance. The ROC curve for the gold standard is the line connecting (0,0) to (0,1) and (0,1) to (1,1).

```
<IPython.core.display.Image object>
```

```
<IPython.core.display.Image object>
```

An animation to demonstrate how an ROC curve relates to sensitivity and specificity for all possible cutoffs (Source)

High Threshold:

- High specificity
- Low sensitivity

Low Threshold

- Low specificity
- High sensitivity

The area under ROC is called *Area Under the Curve(AUC)*. AUC gives the rate of successful classification by the logistic model. To get a more in-depth idea of what a ROC-AUC curve is and how is it calculated, here is a good blog link.

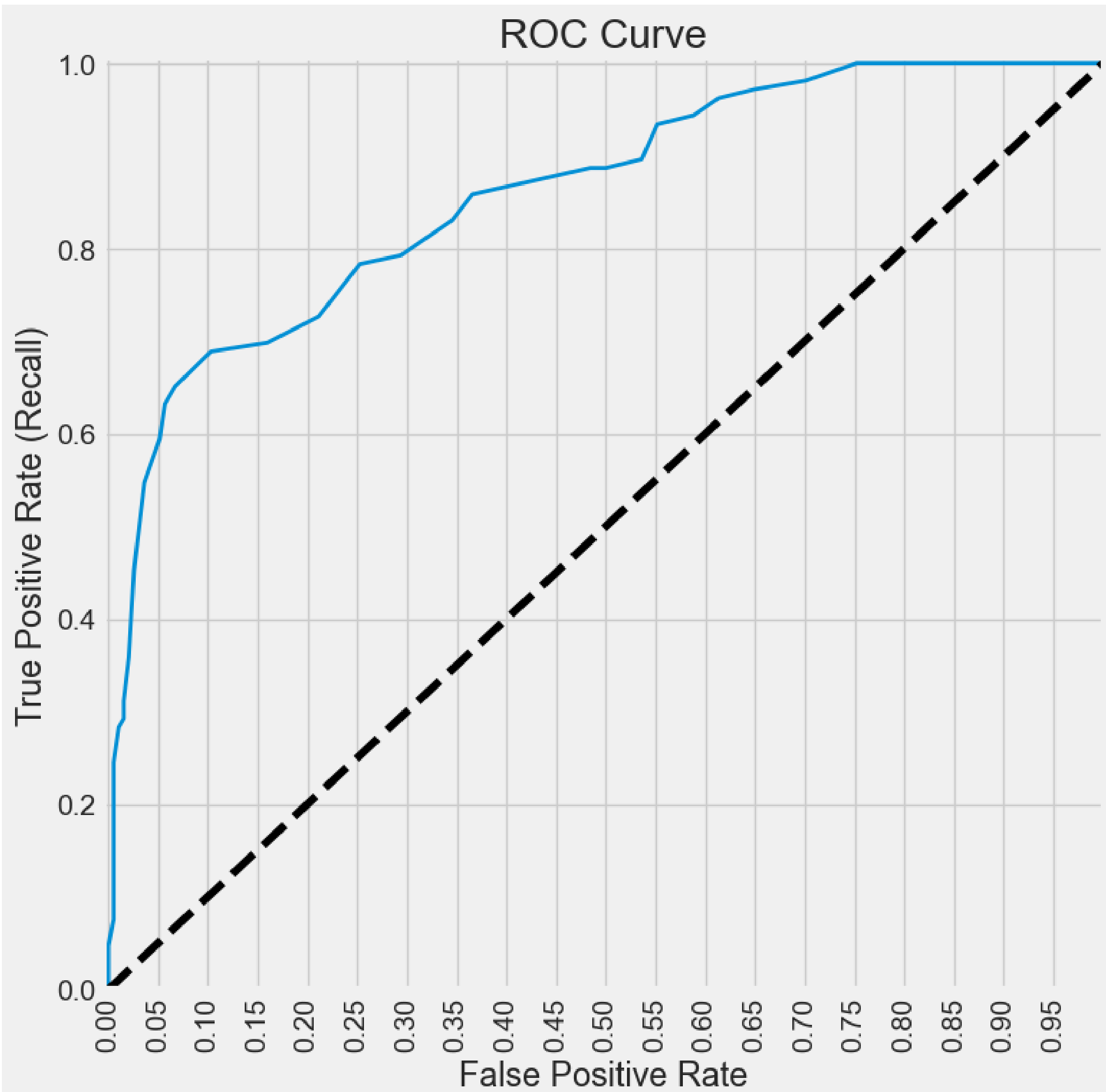
Here is good post by google developers on interpreting ROC-AUC, and its advantages / disadvantages.

Visualize the ROC curve and compute the ROC-AUC for the model `logit_model_diabetes`.

```
y=train.Purchased
ypred = lm.predict(train)
fpr, tpr, auc_thresholds = roc_curve(y, ypred)
print(auc(fpr, tpr))# AUC of ROC
def plot_roc_curve(fpr, tpr, label=None):
    plt.figure(figsize=(8,8))
    plt.title('ROC Curve')
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.005, 1, 0, 1.005])
```

```
    plt.xticks(np.arange(0,1, 0.05), rotation=90)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate (Recall)")

fpr, tpr, auc_thresholds = roc_curve(y, ypred)
plot_roc_curve(fpr, tpr)
```

```
0.8593901964598327
```



Q: How are the values of the `auc_thresholds` chosen to make the ROC curve? Why does it look like a step function?

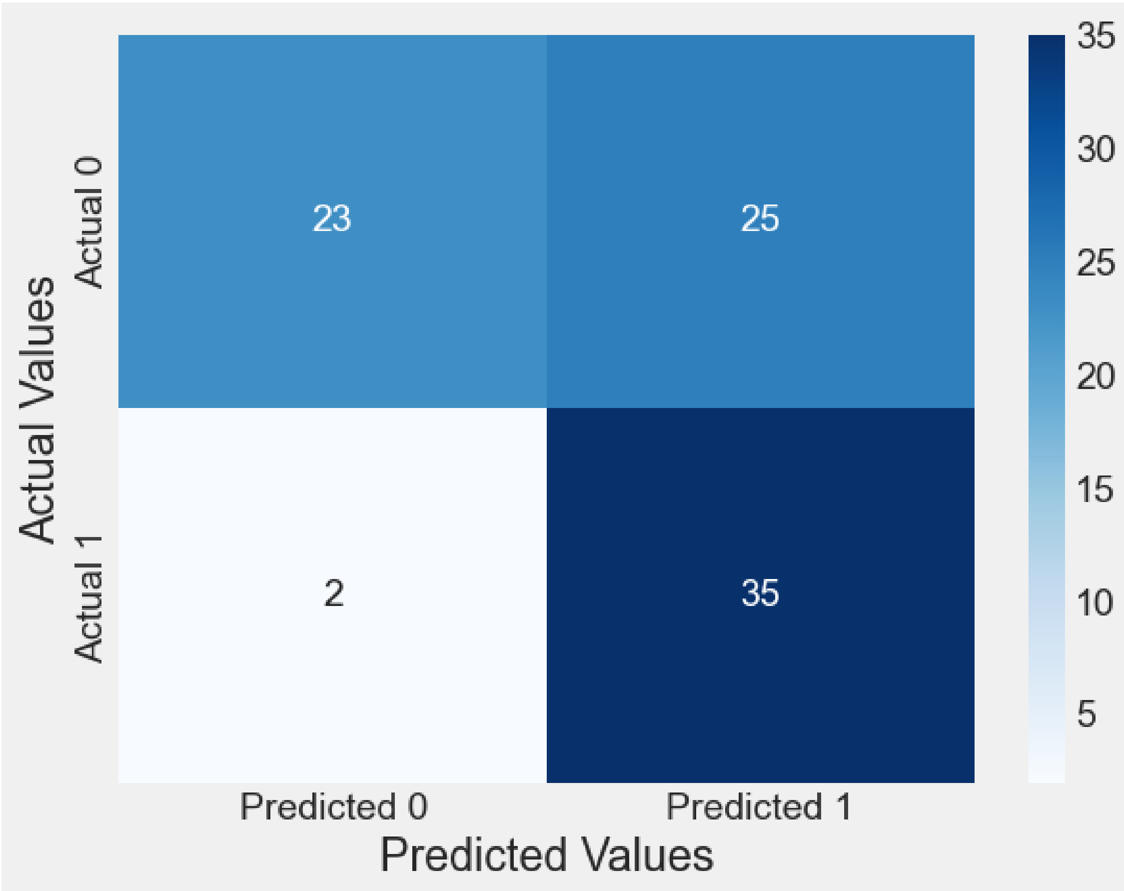
Below is a function that prints the confusion matrix along with all the performance metrics we discussed above for a given decision threshold probability, on train / test data. Note that ROC-AUC does not depend on a decision threshold probability.

```
#Function to compute confusion matrix and prediction accuracy on test/train data
def confusion_matrix_data(data,actual_values,model,cutoff=0.5):
#Predict the values using the Logit model
    pred_values = model.predict(data)
# Specify the bins
    bins=np.array([0,cutoff,1])
#Confusion matrix
    cm = np.histogram2d(actual_values, pred_values, bins=bins)[0]
    cm_df = pd.DataFrame(cm)
    cm_df.columns = ['Predicted 0','Predicted 1']
    cm_df = cm_df.rename(index={0: 'Actual 0',1:'Actual 1'})
# Calculate the accuracy
    accuracy = (cm[0,0]+cm[1,1])/cm.sum()
    fnr = (cm[1,0])/(cm[1,0]+cm[1,1])
    precision = (cm[1,1])/(cm[0,1]+cm[1,1])
    fpr = (cm[0,1])/(cm[0,0]+cm[0,1])
    tpr = (cm[1,1])/(cm[1,0]+cm[1,1])
    fpr_roc, tpr_roc, auc_thresholds = roc_curve(actual_values, pred_values)
    auc_value = (auc(fpr_roc, tpr_roc))# AUC of ROC
    sns.heatmap(cm_df, annot=True, cmap='Blues', fmt='g')
    plt.ylabel("Actual Values")
    plt.xlabel("Predicted Values")
    print("Classification accuracy = {:.1%}".format(accuracy))
    print("Precision = {:.1%}".format(precision))
    print("TPR or Recall = {:.1%}".format(tpr))
    print("FNR = {:.1%}".format(fnr))
    print("FPR = {:.1%}".format(fpr))
    print("ROC-AUC = {:.1%}".format(auc_value))
```

```
confusion_matrix_data(test,test.Purchased,lm,0.3)
```

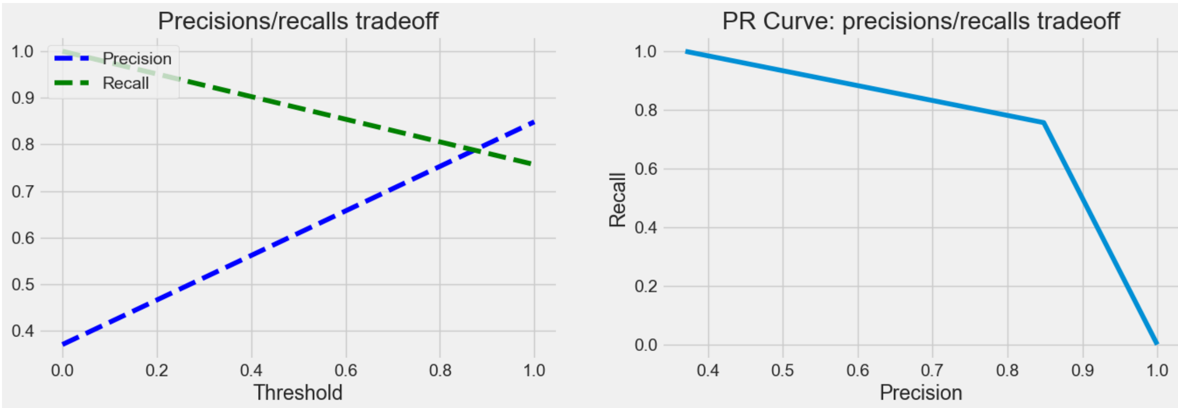
```
Classification accuracy = 68.2%
Precision = 58.3%
TPR or Recall = 94.6%
FNR = 5.4%
```

```
FPR = 52.1%
ROC-AUC = 89.4%
```



8 Precision/Recall Tradeoff

```python
from sklearn.metrics import precision_recall_curve


def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g--", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.title("Precisions/recalls tradeoff")


precisions, recalls, thresholds = precision_recall_curve(y_test, y_pred_test)


plt.figure(figsize=(15, 10))
plt.subplot(2, 2, 1)
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)

plt.subplot(2, 2, 2)
plt.plot(precisions, recalls)
plt.xlabel("Precision")
plt.ylabel("Recall")
plt.title("PR Curve: precisions/recalls tradeoff");
```



8.1 The Receiver Operating Characteristics (ROC) Curve

A **ROC(Receiver Operator Characteristic Curve)** is a plot of sensitivity (True Positive Rate) on the y axis against (1−specificity) (False Positive Rate) on the x axis for varying values of the threshold t. The 45° diagonal line connecting (0,0) to (1,1) is the ROC curve corresponding to random chance. The ROC curve for the gold standard is the line connecting (0,0) to (0,1) and (0,1) to (1,1).