Data Science II with python (Class notes)

STAT 303-2

Arvind Krishna

2023-01-03

Table of contents

Preface

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

The course notes are currently being written, and will continue to being developed as the course progresses (just like the course textbook last quarter). Please report any typos / mistakes / inconsistencies / issues with the class notes / class presentations in your comments here. Thank you!

Part I

Linear regression

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
```

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
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

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
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

```
# Let's create the model

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

# 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:	Mon, 08 Jan 2024	**Prob (F-statistic):**	0.00
Time:	10:06:15	**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: car price = -4122.0357 + 12990 * `engineSize`

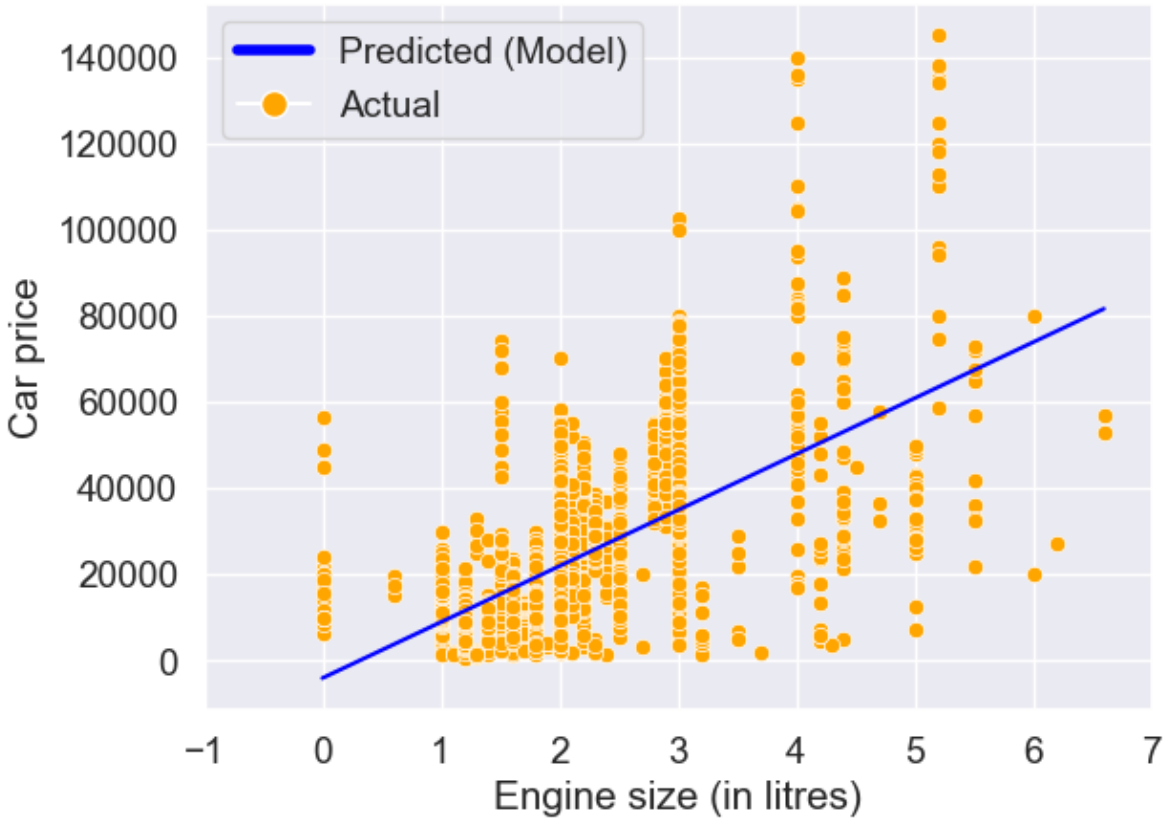
The coefficients can also be returned directly usign the `params` attribute of the `model` object returned by the `fit()` method of the `ols` class:

```
model.params
```

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

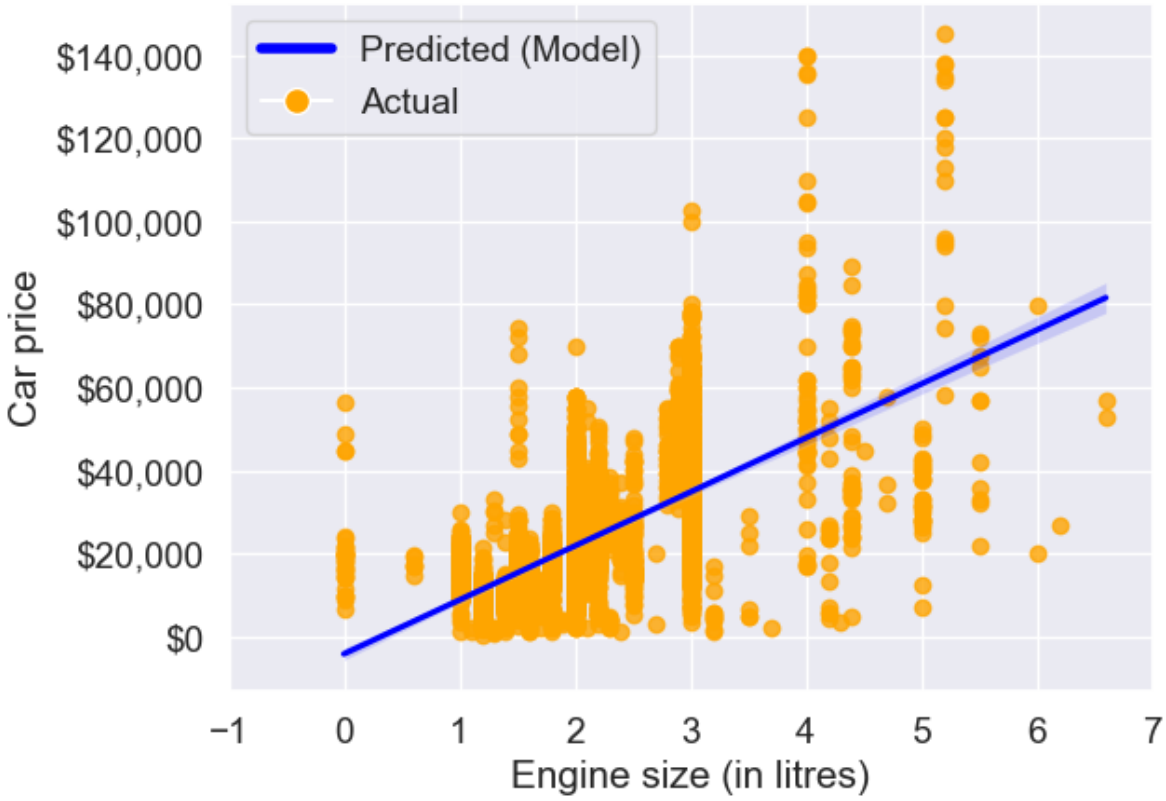
Visualize the regression line

```
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={"
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
```



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*

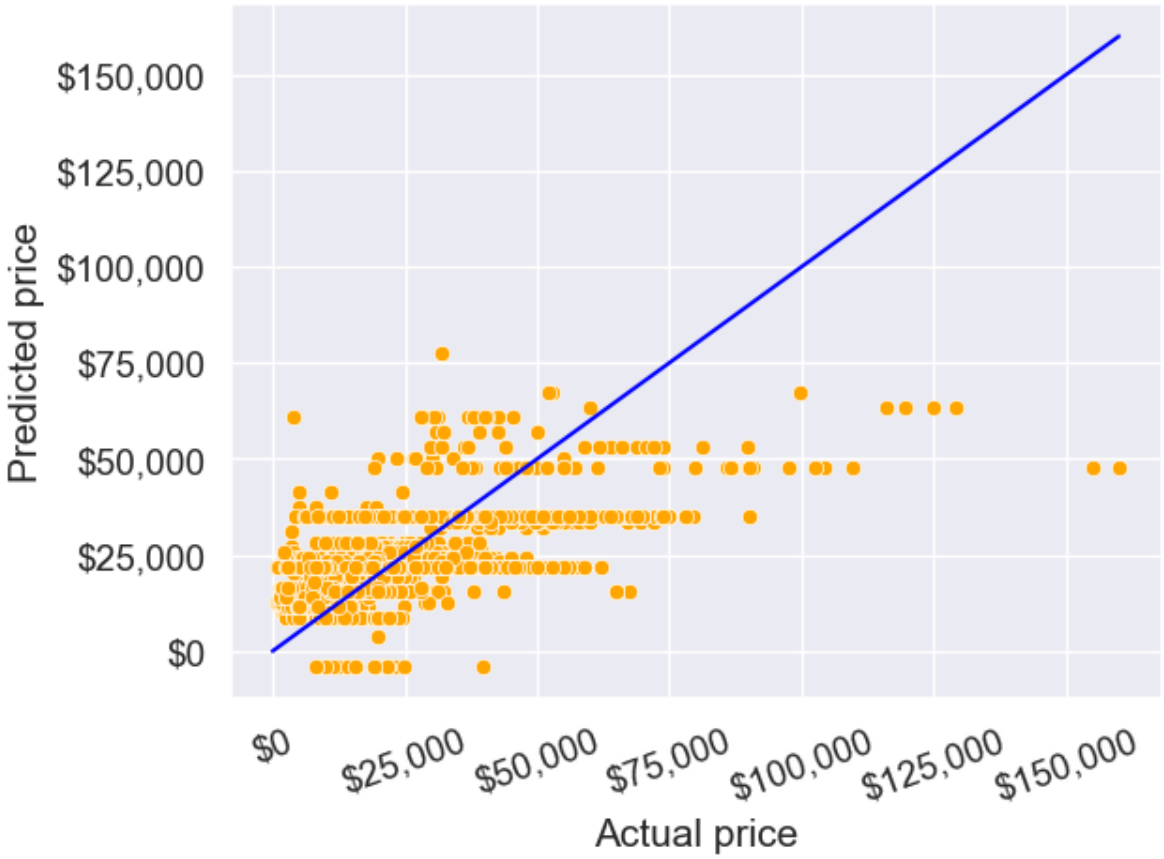
Now that the model has been trained, let us evaluate it on unseen data. Make sure that the columns names of the predictors are the same in train and test datasets.

```
# 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 ca
pred_price = model.predict(testf)#Note that the predict() function finds the predictor 'en
```

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') #Plot
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())
```

```
12995.1064515487
```

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)
```

```
12810.109175214136
```

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 interva
intervals = model.get_prediction(testf)
```

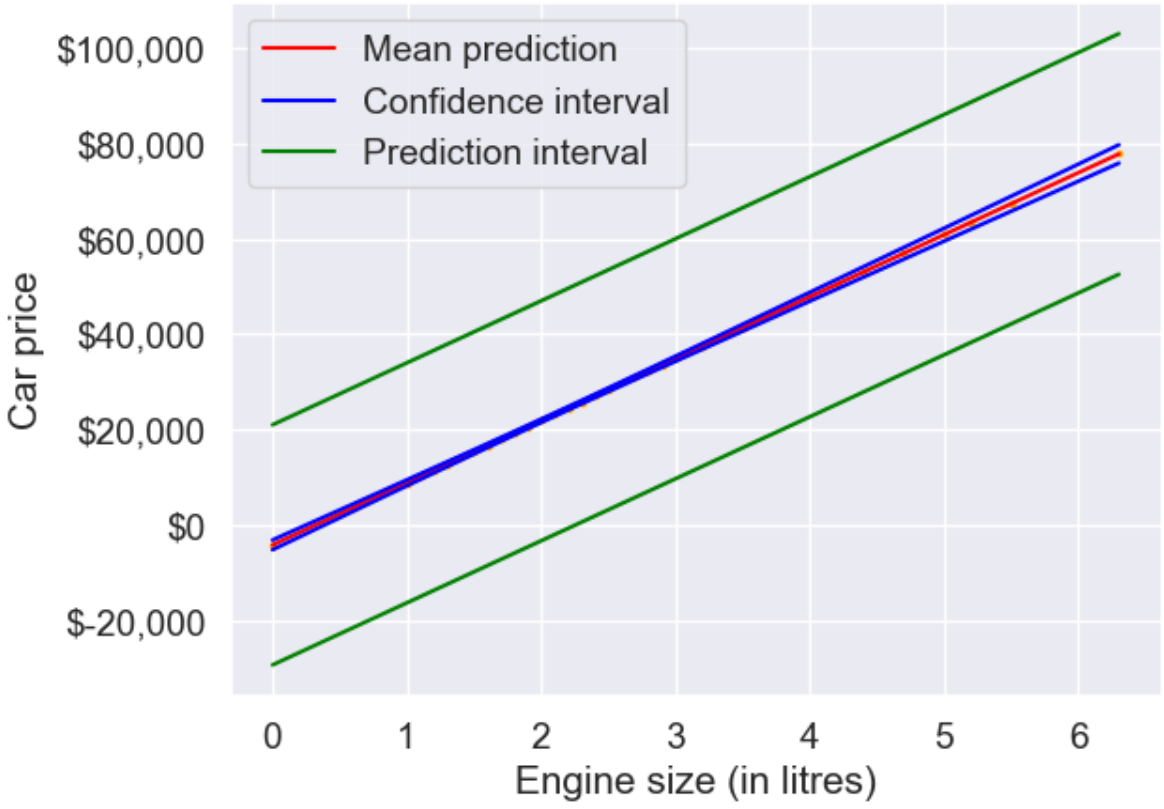
```
#The function requires specifying alpha (probability of Type 1 error) instead of the confi
intervals.summary_frame(alpha=0.05)
```

	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
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
    # 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]
```

```
# 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
```

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 model is similar to that of `sklearn`'s `LinearRegression()` function. However, the order in which the predictors and response is specified is different. The formula-style syntax 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)

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

# 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.

```
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

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

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*

```
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
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
#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()
```

Table 2.2: 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, 27 Dec 2022	Prob (F-statistic):	0.00
Time:	01:07:25	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

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

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

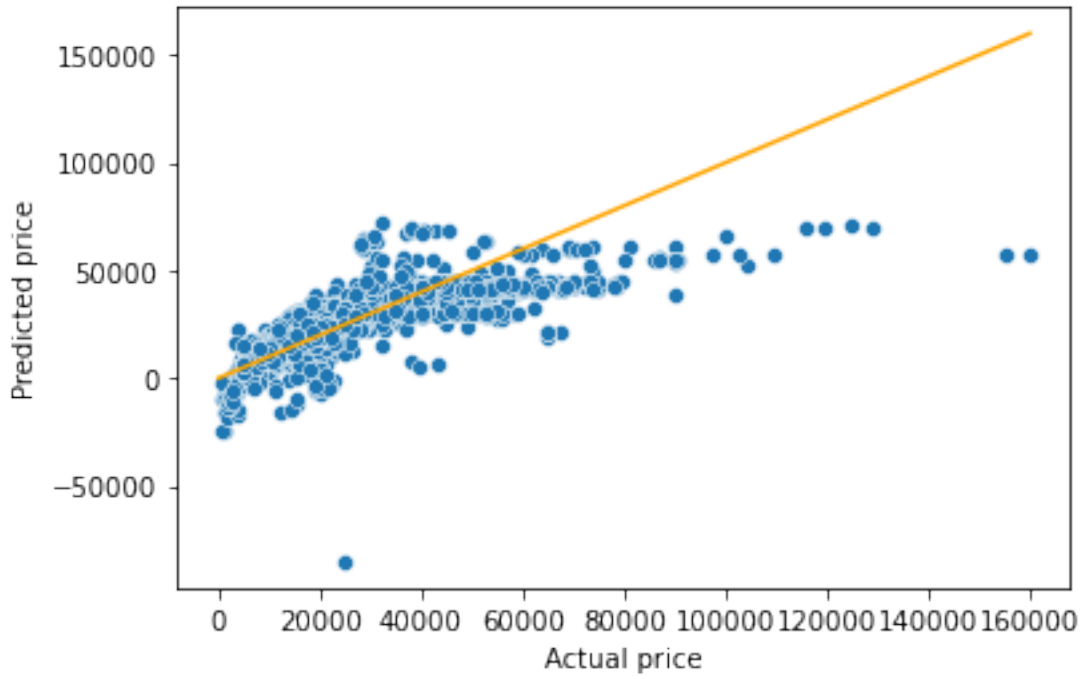
```
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
```

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

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

```
sns.scatterplot(x = testp.price, y = pred_price)
#In case of a perfect prediction, all the points must lie on the line x = y.
sns.lineplot(x = [0,testp.price.max()], y = [0,testp.price.max()],color='orange') #Plottin
plt.xlabel('Actual price')
plt.ylabel('Predicted price')
```

```
Text(0, 0.5, 'Predicted price')
```



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 R2 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
```

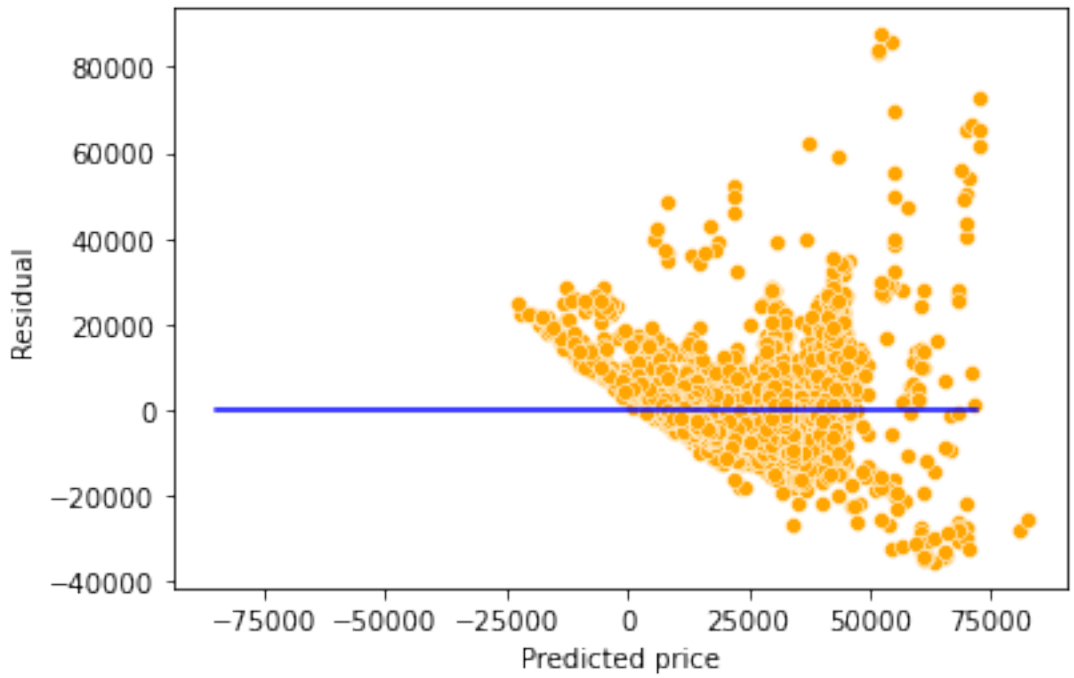
What is the residual standard error based on the training data?

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

```
9563.74782917604
```

```
sns.scatterplot(x = model.fittedvalues, y=model.resid,color = 'orange')
sns.lineplot(x = [pred_price.min(),pred_price.max()],y = [0,0],color = 'blue')
plt.xlabel('Predicted price')
plt.ylabel('Residual')
```

```
Text(0, 0.5, 'Residual')
```



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 2.5: 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-squared). 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-squared.

3 Variable interactions and transformations

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
```

```
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

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:

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

or

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

or

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

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:

$$price = 5.606e5 - 275.3833 * year + (-1.796e6 + 896.7687 * year) * engineSize - 0.1525 * mileage - 84.3417 * mpg \quad (3.4)$$

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 $\approx$ \$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 $\approx$ \$15,500 , assuming all the other predictors are constant.

Similarly, the equation can be re-arranged as:

$$price = 5.606e5 + (-275.3833 + 896.7687 * engineSize) * year - 1.796e6 * engineSize - 0.1525 * mileage - 84.3417 * mp \quad (3.5)$$

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 $\approx$ \$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

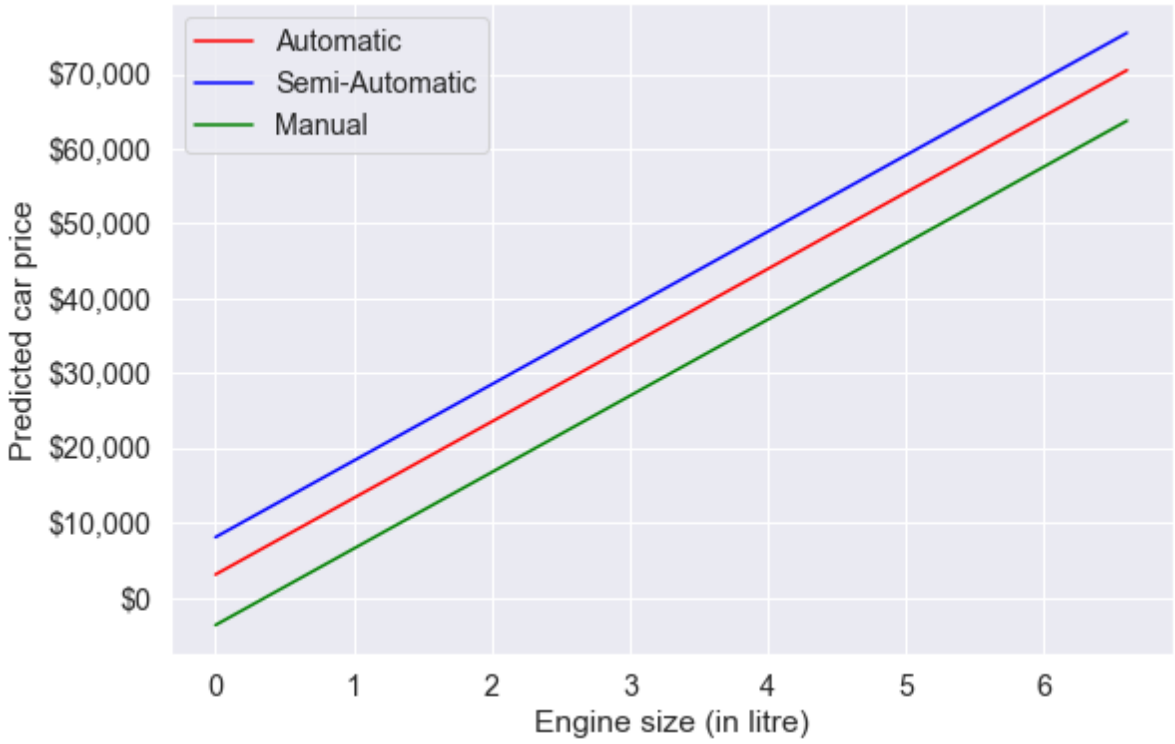
Ans: For the hypothetical scenario of a car with zero engine size and *Automatic* transmission, the estimated mean car price is $\approx$ \$3042.

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.param
sns.lineplot(x = x, y = model.params['engineSize']*x+model.params['Intercept']+model.param
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"))', da
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: $price = 5523 + 4642 * engineSize$,

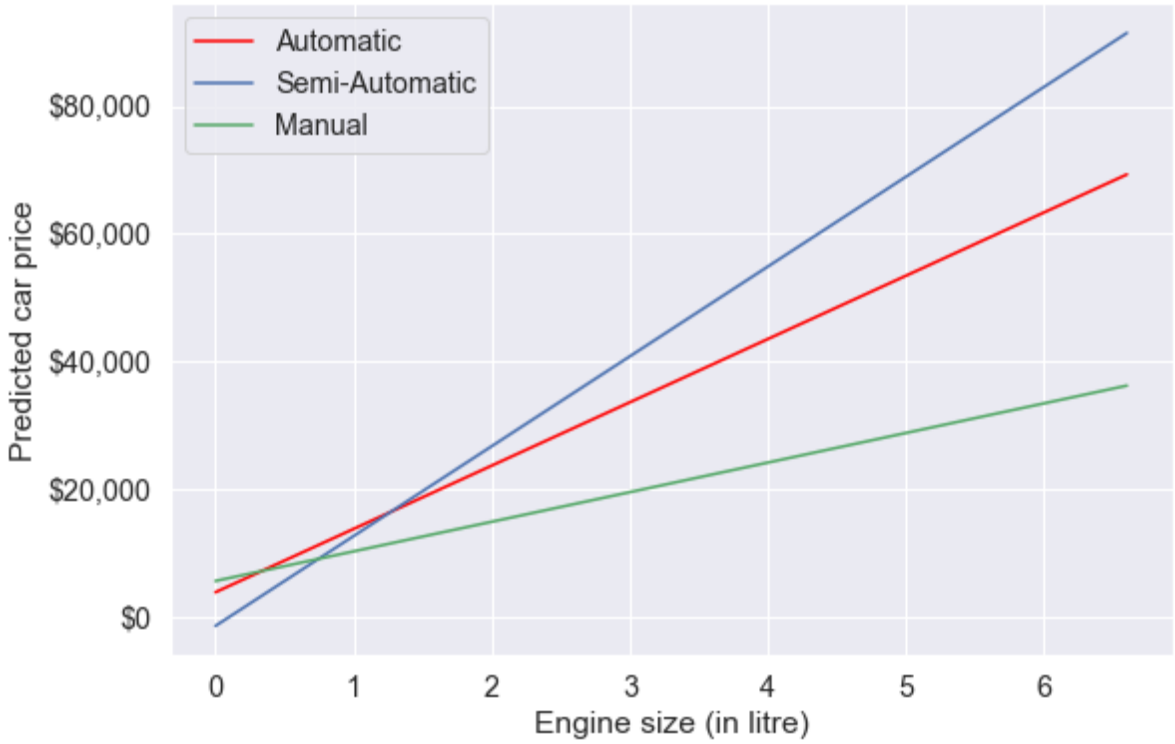
Q: Interpret the coefficient of manual tranmission, i.e., the coefficient of `transmission[T.Manual]`.

A: For a given engine size, the estimated mean `price` of a car with *Manual* transmission is $\approx$ \$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 $\approx$ \$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
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]'])
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.

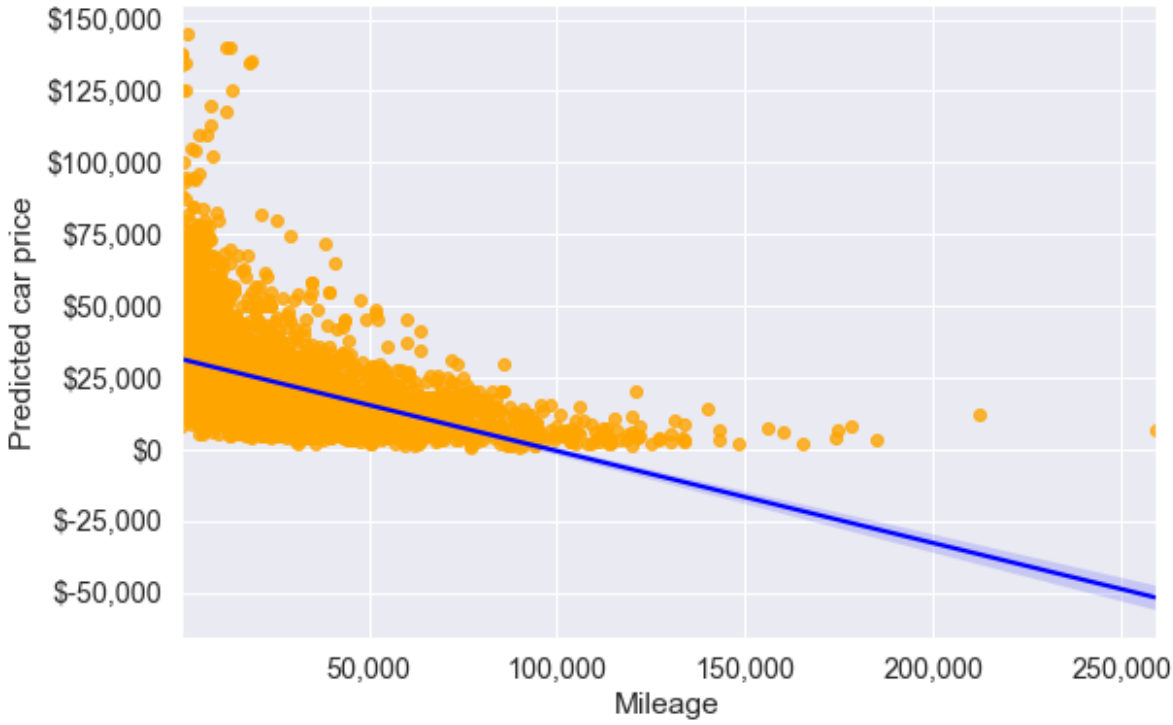
The slope of the regression line for semi-automatic cars is the largest. This suggests that increase in engine size is associated with a higher increase in car price for semi-automatic cars, as compared to other cars.

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_
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

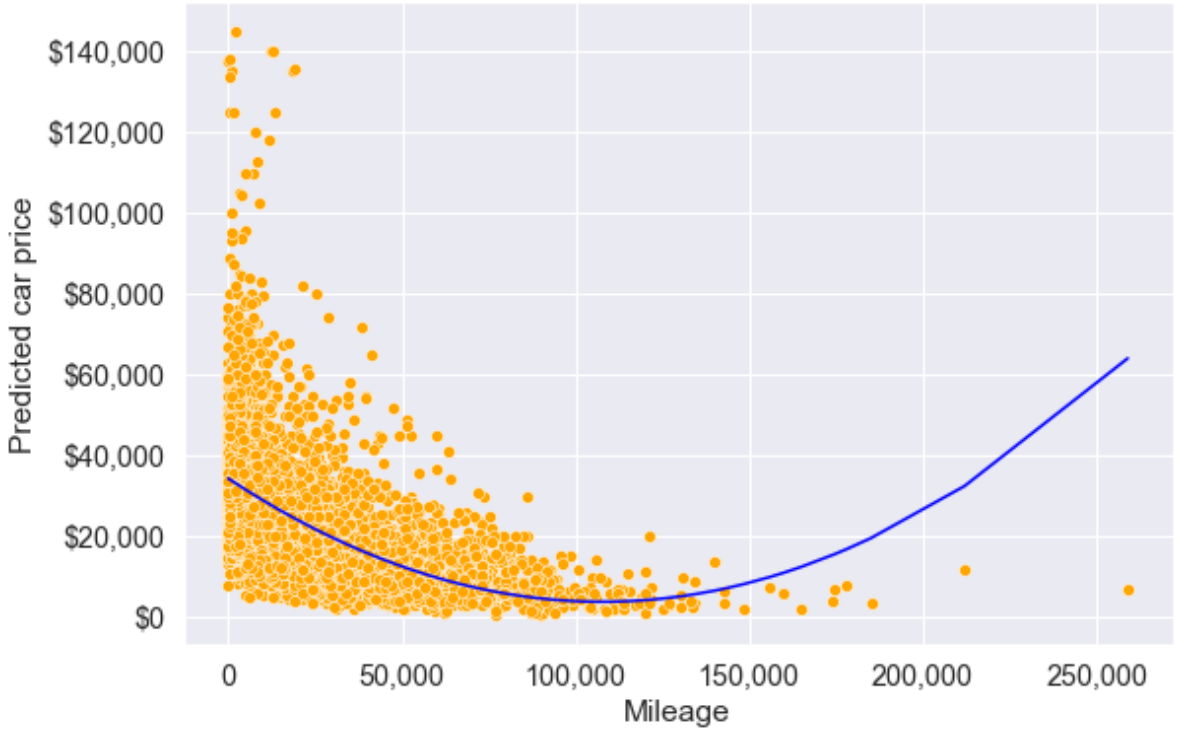
Omnibus:	2362.973	Durbin-Watson:	0.325
Prob(Omnibus):	0.000	Jarque-Bera (JB):	22427.952
Skew:	2.052	Prob(JB):	0.00
Kurtosis:	12.576	Cond. No.	4.81e+09

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
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_u
model = ols_object.fit()
model.summary()
```

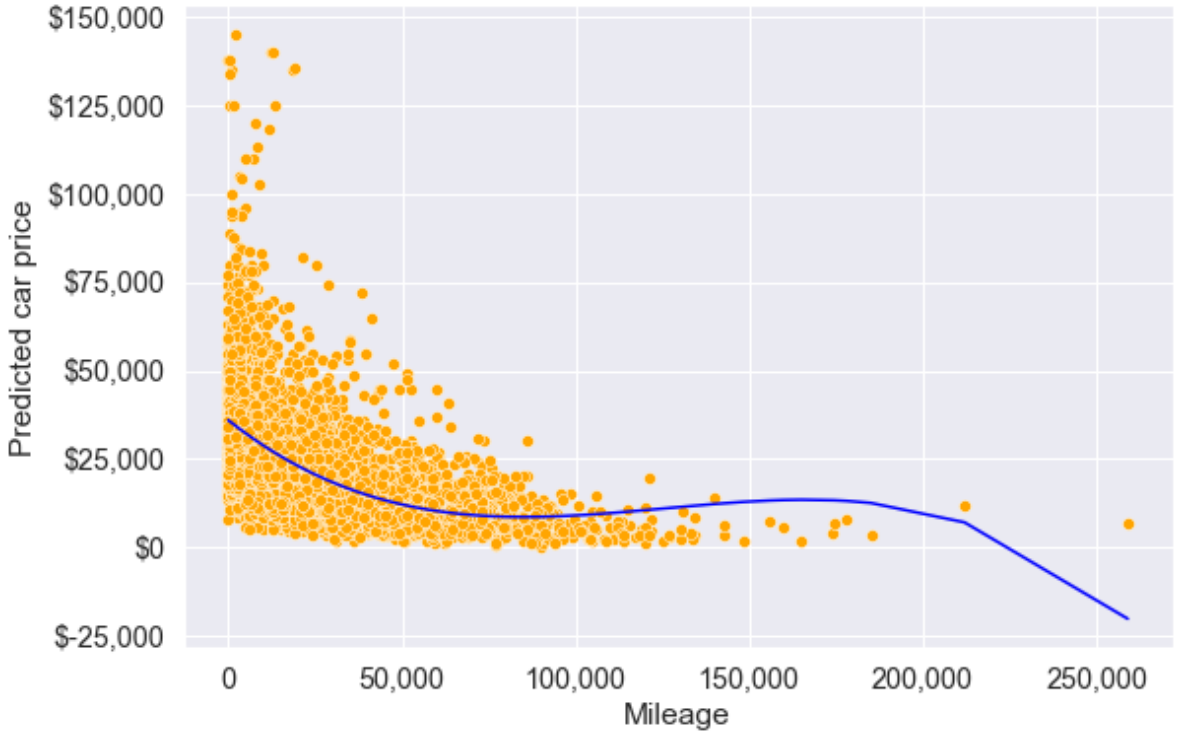
Table 3.17: OLS Regression Results

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 = t
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.

4 Model assumptions

Read section 3.3.3 (1 & 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.

Consider the model with interactions and transformation developed previously.

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

```
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

```
ols_object = smf.ols(formula = 'price~(year+engineSize+mileage+mpg)**2+I(mileage**2)', dat
model = ols_object.fit()
model.summary()
```

Table 4.2: OLS Regression Results

Dep. Variable:	price	R-squared:	0.732
Model:	OLS	Adj. R-squared:	0.731
Method:	Least Squares	F-statistic:	1229.
Date:	Wed, 25 Jan 2023	Prob (F-statistic):	0.00
Time:	11:36:00	Log-Likelihood:	-51911.
No. Observations:	4960	AIC:	1.038e+05
Df Residuals:	4948	BIC:	1.039e+05
Df Model:	11		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-1.282e+06	7.14e+05	-1.795	0.073	-2.68e+06	1.18e+05
year	632.3954	353.865	1.787	0.074	-61.338	1326.128
engineSize	-1.465e+06	1.61e+05	-9.129	0.000	-1.78e+06	-1.15e+06
mileage	56.4581	3.811	14.815	0.000	48.987	63.929
mpg	-2.951e+04	9550.775	-3.089	0.002	-4.82e+04	-1.08e+04
year:engineSize	735.8074	79.532	9.252	0.000	579.890	891.725
year:mileage	-0.0281	0.002	-14.898	0.000	-0.032	-0.024
year:mpg	14.6915	4.731	3.105	0.002	5.417	23.966
engineSize:mileage	-0.0808	0.011	-7.143	0.000	-0.103	-0.059
engineSize:mpg	-120.5780	11.384	-10.592	0.000	-142.896	-98.260
mileage:mpg	0.0026	0.000	5.173	0.000	0.002	0.004
I(mileage ** 2)	3.495e-07	1.56e-07	2.236	0.025	4.31e-08	6.56e-07

Omnibus:	1958.631	Durbin-Watson:	0.542
Prob(Omnibus):	0.000	Jarque-Bera (JB):	44560.042
Skew:	1.349	Prob(JB):	0.00
Kurtosis:	17.434	Cond. No.	1.73e+13

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

```
8502.851955843495
```

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

```
8708.676318160937
```

```
#Computing MAE on test data
pred_price = model.predict(testf)
(np.abs(testp.price - pred_price)).mean()
```

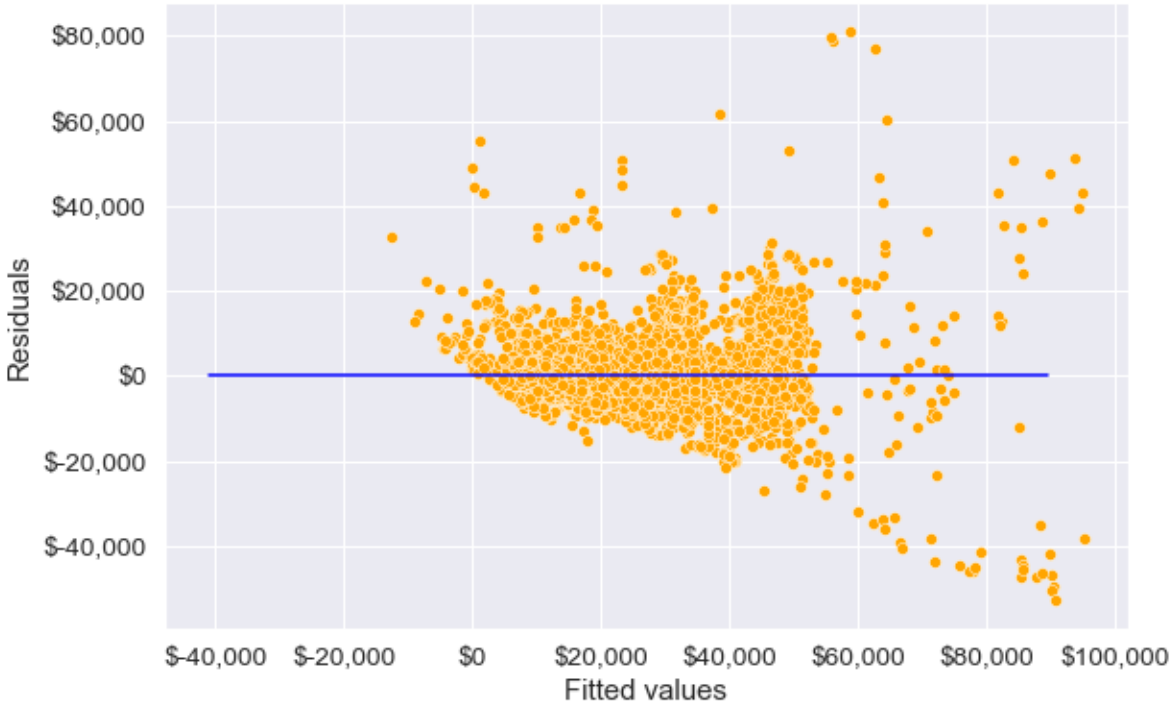
```
5395.006622253402
```

Let us check if this model satisfies the assumptions of the linear regression model

4.1 Non-linearity of data

We have assumed that there is a linear relationship between the predictors and the response. Residual plots, which are scatter plots of residuals vs fitted values, can be used to identify non-linearity. Fitted values are the values estimated by the model on training data, denoted by $\hat{y}_i$, and residuals are given by $e_i = y_i - \hat{y}_i$.

```
#Plotting residuals vs fitted values
plt.rcParams["figure.figsize"] = (9,6)
sns.set(font_scale=1.25)
ax = sns.scatterplot(x = model.fittedvalues, y=model.resid,color = 'orange')
sns.lineplot(x = [pred_price.min(),pred_price.max()],y = [0,0],color = 'blue')
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('${x:,.0f}')
```



The model seems to satisfy this assumption, as we do not observe a strong pattern in the residuals around the line Residuals = 0. Residuals are distributed more or less in a similar manner on both sides of the blue line for all fitted values.

For the model to satisfy the linearity assumption perfectly, the points above the line *(Residuals = 0)*, should be mirror image of the points below the line, i.e., the blue line in the above plot should act as a mirror.

What to do if there is non-linear association (page 94 of book): If the residual plot indicates that there are non-linear associations in the data, then a simple approach is to use non-linear transformations of the predictors, such as $logX, \sqrt{X}$, and X^2, in the regression model.

4.2 Non-constant variance of error terms

The variance of the error terms is assumed to be constant, i.e., $Var(\epsilon_i) = \sigma^2$, and this assumption is used while deriving the standard errors of the regression coefficients. The standard errors in turn are used to test the significant of the predictors, and obtain their confidence interval. Thus, violation of this assumption may lead to incorrect inference. Non-constant variance of error terms, or violation of the constant variance assumption, is called *heteroscedasticity.*

This assumption can be checked by plotting the residuals against fitted values.

```
#Plotting residuals vs fitted values
ax = sns.scatterplot(x = model.fittedvalues, y=model.resid,color = 'orange')
sns.lineplot(x = [pred_price.min(),pred_price.max()],y = [0,0],color = 'blue')
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
ax.yaxis.set_major_formatter('${x:,.0f}')
ax.xaxis.set_major_formatter('${x:,.0f}')
```

We see that the variance of errors seems to increase with increase in the fitted values. In such a case a log transformation of the response can resolve the issue to some extent. This is because a log transformation will result in a higher shrinkage of larger values.

```
#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**
model_log = ols_object.fit()
model_log.summary()
```

Table 4.5: OLS Regression Results

Dep. Variable:	np.log(price)	R-squared:	0.803
Model:	OLS	Adj. R-squared:	0.803
Method:	Least Squares	F-statistic:	1834.
Date:	Wed, 25 Jan 2023	Prob (F-statistic):	0.00
Time:	11:37:55	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:	711.515	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

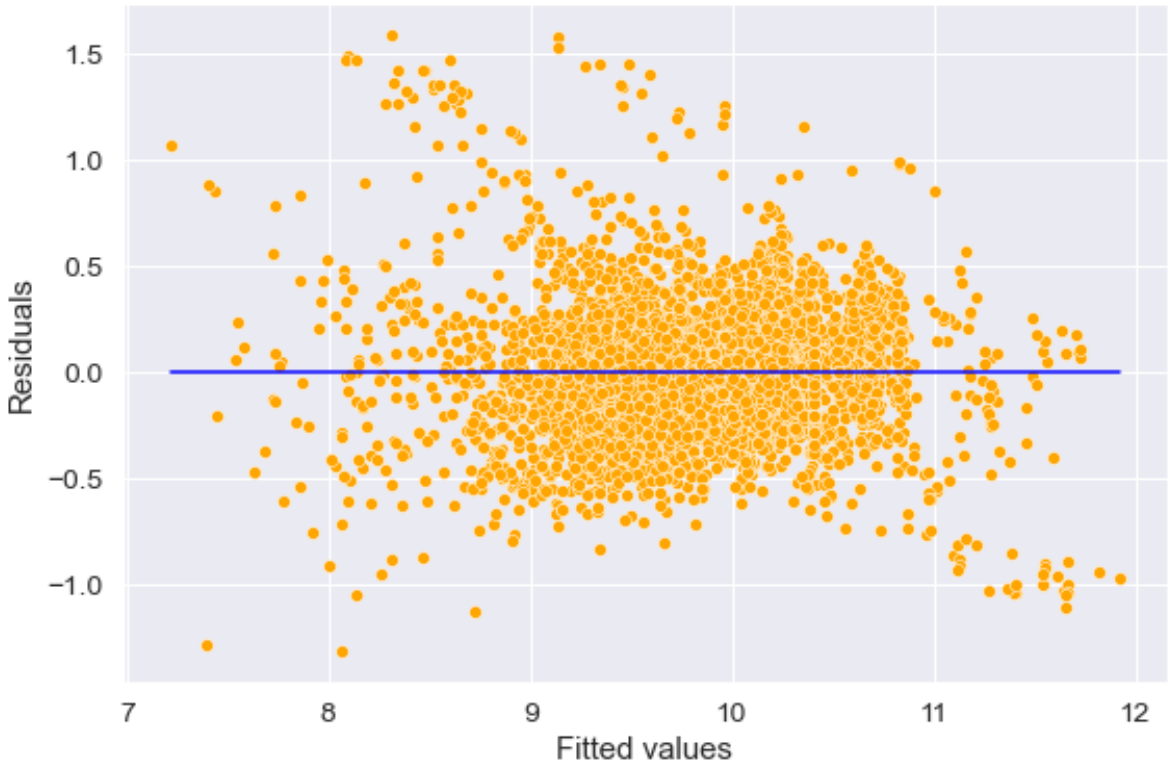
Note that the coefficient of *year* turns out to be significant (at 5% significance level), unlike in the previous model. Intuitively, the coefficient of *year* should have been significant, as *year* has the highest linear correlation of 50% with car *price*.

Although the R-squared has increased as compared to the previous model, violation of this assumption does not cause bias in the regression coefficients. Thus, there may not be a large improvement in the model fit, unless we add predictor(s) to address heteroscedasticity.

Let us check the constant variance assumption again.

```
#Plotting residuals vs fitted values
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],col
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
```

```
Text(0, 0.5, 'Residuals')
```



Now we observe that the constant variance assumption is satisfied. Let us see the RMSE of this model on test data.

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

```
9094.209503063496
```

Note that the RMSE of the log-transformed model has increased as compared to the model without transformation. Does it mean the log-transformed model is less accurate?

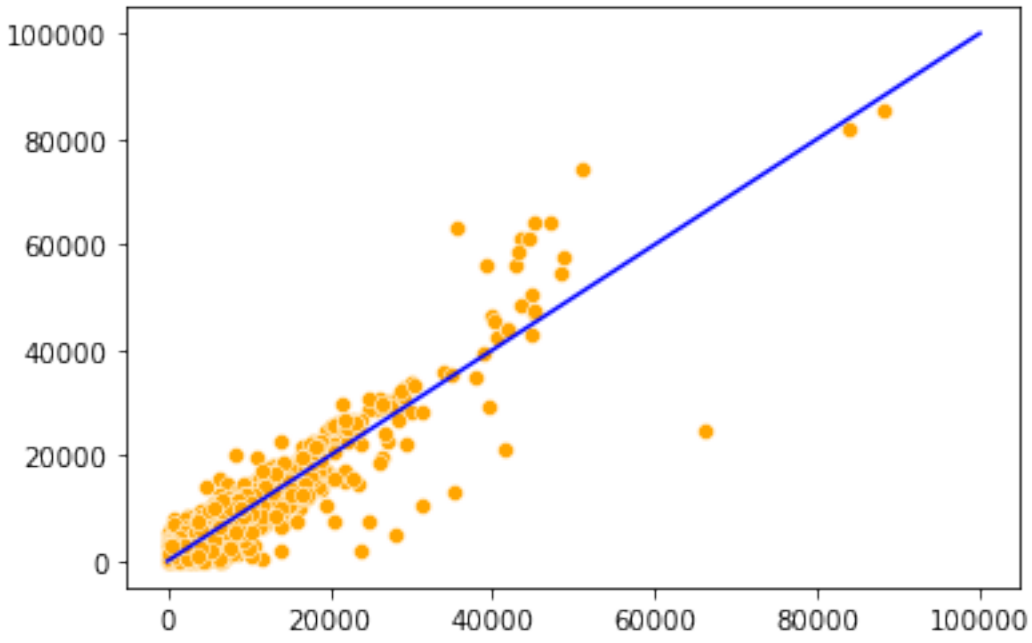
```
#Computing MAE on test data
pred_price_log = model_log.predict(testf)
((np.abs(testp.price - np.exp(pred_price_log))).mean())
```

```
5268.398904745121
```

Although the RMSE has increased a bit for the log-transformed model, the MAE has reduced. This means the log-transformed model does a bit worse on reducing relatively large errors, but does better in reducing the absolute errors on an average.

```
#Comparing errors of the log-transformed model with the previous model
err = np.abs(testp.price - pred_price)
err_log = np.abs(testp.price - np.exp(pred_price_log))
sns.scatterplot(x = err,y = err_log, color = 'orange')
sns.lineplot(x = [0,100000], y = [0,100000], color = 'blue')
np.sum(err_log<err)/len(err)
```

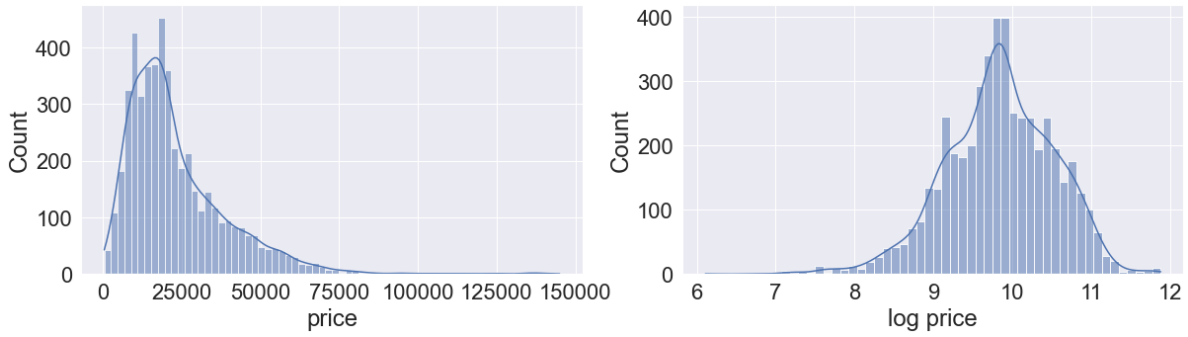
```
0.5572604790419161
```



For 56% of the cars, the log transformed makes a more accurate prediction than the previous model, which is another criterion based on which the log-transformed model is more accurate. However, the conclusion based on RMSE is different. This is because RMSE can be influenced by a few large errors. Thus, RMSE, though sometimes appropriate than other criteria, should not be used as the sole measure to compare the accuracy of models.

```
#Visualizing the distribution of price and log(price)
fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.2)
sns.set(rc = {'figure.figsize':(20,12)})
sns.set(font_scale = 2)
ax = fig.add_subplot(2, 2, 1)
sns.histplot(train.price,kde=True)
ax.set(xlabel='price', ylabel='Count')
ax = fig.add_subplot(2, 2, 2)
sns.histplot(np.log(train.price),kde=True)
ax.set(xlabel='log price', ylabel='Count')
```

```
[Text(0.5, 0, 'log price'), Text(0, 0.5, 'Count')]
```



We can see that the log transformation shrinked the higher values of price, making its distribution closer to normal.

Note that heteroscedasticity can also occur due to model misspecification, i.e., in case of missing predictor(s). Some of the cars are too expensive, which makes the `price` distribution skewed. Perhaps, the price of expensive cars could be better explained by the car `model`, a predictor that is missing in the current model.

5 Potential issues

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
```

```
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**
model_log = ols_object.fit()
```

```
model_log.summary()
```

Table 5.2: OLS Regression Results

Dep. Variable:	np.log(price)	R-squared:	0.803
Model:	OLS	Adj. R-squared:	0.803
Method:	Least Squares	F-statistic:	1834.
Date:	Sun, 05 Feb 2023	Prob (F-statistic):	0.00
Time:	19:31:46	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:	711.515	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

5.1 Outliers

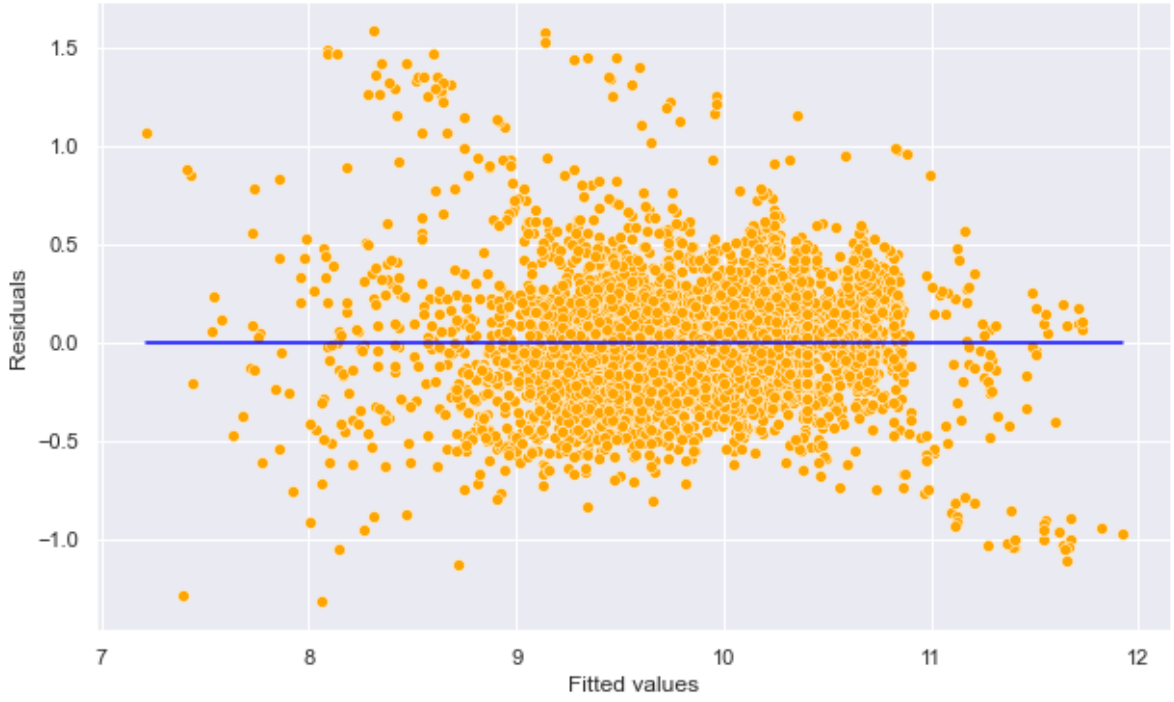
An outlier is a point for which the true response (y_i) is far from the value predicted by the model. Residual plots can be used to identify outliers.

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],col
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
```

```
Text(0, 0.5, '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.

Studentized residuals follow a t distribution with (n–p–2) degrees of freedom. Thus, in general, observations whose studentized residuals have a magnitude higher than 3 are potential outliers.

Let us find the studentized residuals in our car `price` prediction model.

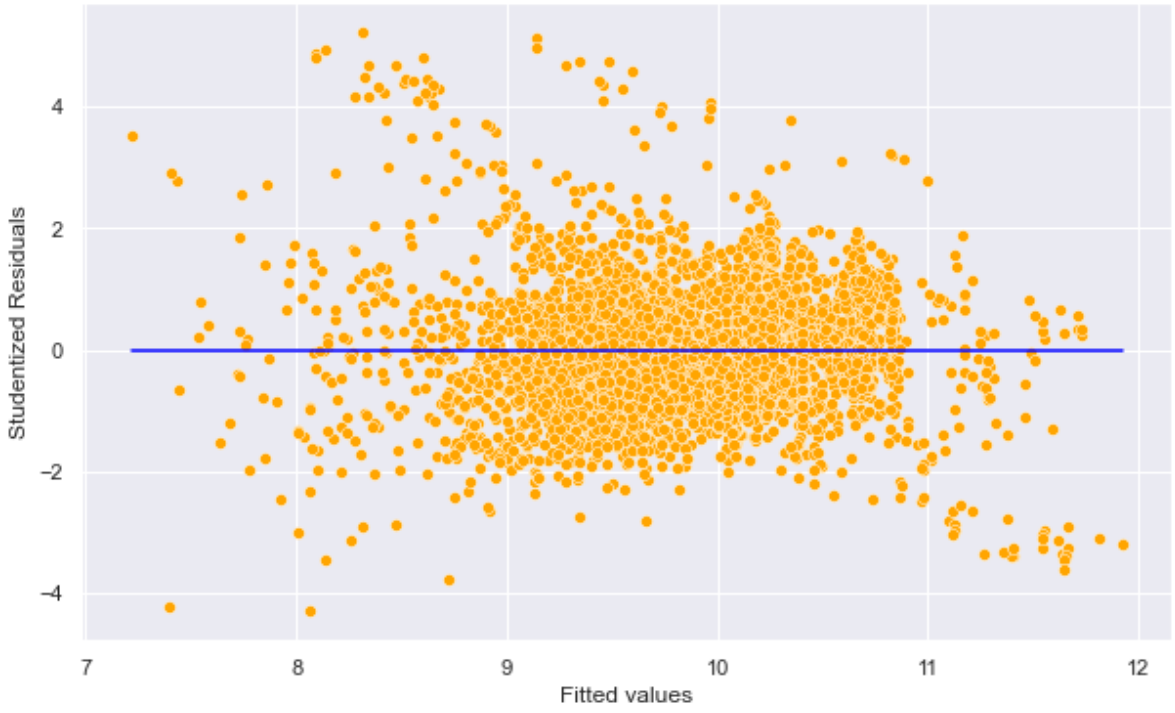
```
#Studentized residuals
out = model_log.outlier_test()
out
```

	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	0.539130	1.0
4	0.027930	0.977719	1.0
...	...	...	...
4955	-0.523361	0.600747	1.0
4956	-0.509539	0.610397	1.0
4957	-1.718802	0.085713	1.0
4958	-0.077595	0.938153	1.0
4959	-0.482388	0.629551	1.0

Studentized residuals are in the first column of the above table.

```
#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],col
plt.xlabel('Fitted values')
plt.ylabel('Studentized Residuals')
```

```
Text(0, 0.5, 'Studentized Residuals')
```



Potential outliers: Observations whose studentized residuals have a magnitude greater than 3.

Impact of outliers: Outliers do not have a large impact on the OLS line / plane / hyperplane. 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.

```
#Number of points with absolute studentized residuals greater than 3
np.sum((np.abs(out.student_resid)>3))
```

```
86
```

Are there outliers in our example?: In the above plot, there are 86 points with absolute studentized residuals larger than 3. However, most of the predictors are significant and R-squared has a relatively high value of 80%. Thus, even if there are outliers, there is no need to remove them as it is unlikely to change the significance of individual variables. Furthermore, looking into the data, we find that the price of some of the luxury cars such as Mercedez G-class is actually much higher than average. So, the potential outliers in the data do not seem to be due to incorrect data. The high studentized residuals may be due to some deficiency in the model, such as missing predictor(s) (like car `model`), rather than incorrect data. Thus, we should not remove any data that has an outlying value of *log(price)*.

Since `model` seems to be a variable that can explain the price of overly expensive cars, let us include it in the regression model.

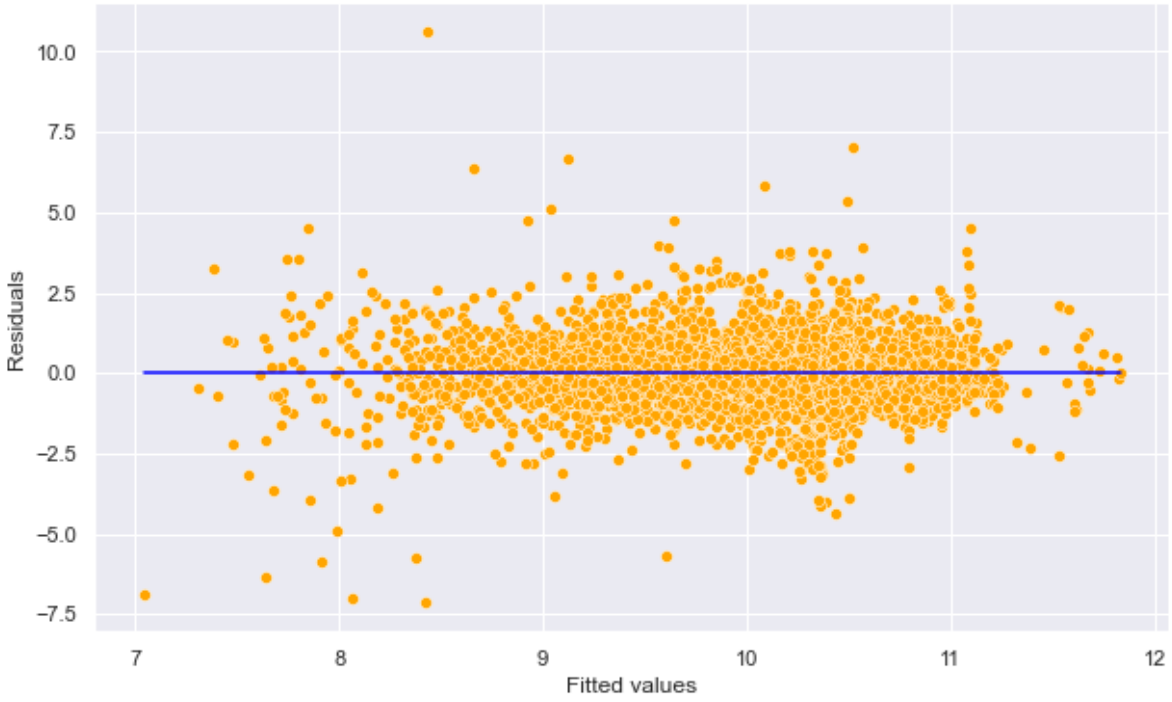
```
#Model with an interaction term and a variable transformation term
ols_object = smf.ols(formula = 'np.log(price)~(year+engineSize+mileage+mpg)**2+I(mileage**
model_log = ols_object.fit()
#Model summary not printed to save space
#model_log.summary()
```

```
#Computing RMSE on test data with car 'model' as one of the predictors
pred_price_log2 = model_log.predict(testf)
np.sqrt(((testp.price - np.exp(pred_price_log2))**2).mean())
```

```
4252.20045604376
```

```
#Plotting studentized residuals vs fitted values for the model with car 'model' as one of
out = model_log.outlier_test()
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],col
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
```

```
Text(0, 0.5, 'Residuals')
```




```
#Number of points with absolute studentized residuals greater than 3
np.sum((np.abs(out.student_resid)>3))
```

```
69
```

Note the RMSE has reduced to almost half of its value as compared to the regression model without the predictor - `model`. Car model does help better explain the variation in price of cars! The number of points with absolute studentized residuals greater than 3 has also reduced to 69 from 86.

5.2 High leverage points

High leverage points are those with an unsual value of the predictor(s). They 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,

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{i'=1}^{n}(x_{i'} - \bar{x})^2}. \tag{5.1}$$

It is clear from this equation that h_i increases with the distance of x_i from $\bar{x}$.The leverage statistic h_i is always between $1/n$ and 1, and the average leverage for all the observations is always equal to $(p+1)/n$. 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.

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 without the predictor - `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**
model_log = ols_object.fit()
model_log.summary()
```

Table 5.6: OLS Regression Results

Dep. Variable:	np.log(price)	R-squared:	0.803
Model:	OLS	Adj. R-squared:	0.803
Method:	Least Squares	F-statistic:	1834.
Date:	Sun, 05 Feb 2023	Prob (F-statistic):	0.00
Time:	19:31:59	Log-Likelihood:	-1173.8
No. Observations:	4960	AIC:	2372.
Df Residuals:	4948	BIC:	2450.
Df Model:	11		
Covariance Type:	nonrobust		

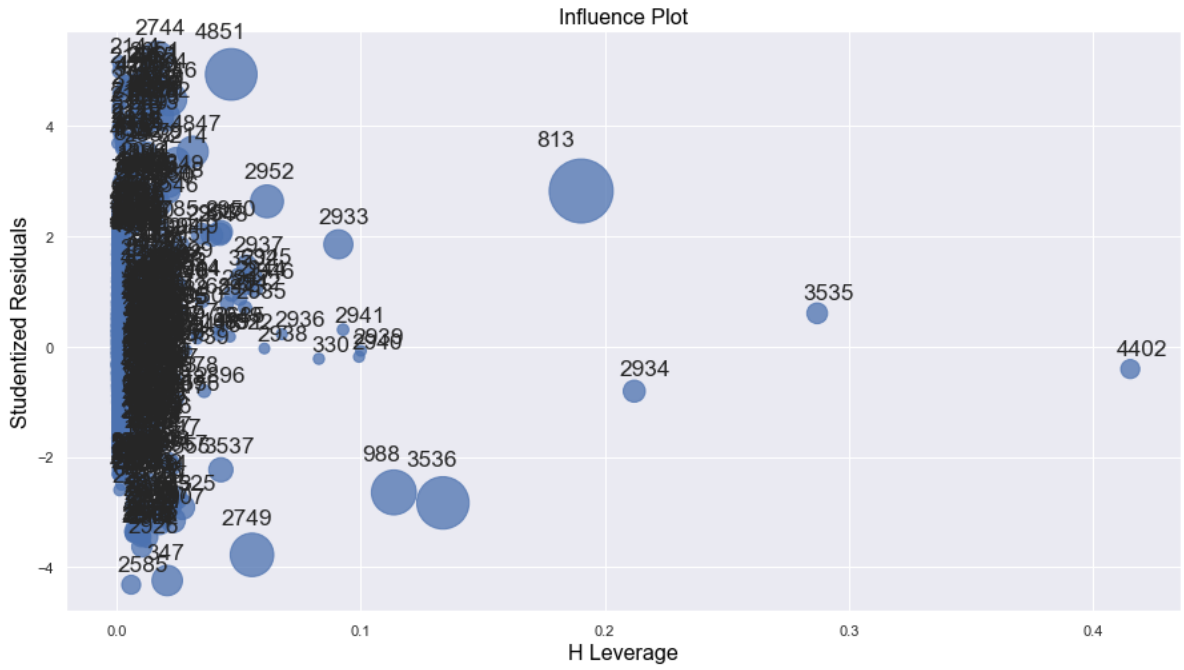
	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-238.2125	25.790	-9.237	0.000	-288.773	-187.652

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:	711.515	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

```
#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 = 4*average_leverage
```

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

```
197
```

5.3 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)>3)[0],
                                           (np.where(leverage>high_leverage_threshold)[0])
```

```
train_filtered.shape
```

```
(4921, 11)
```

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

```
39
```

We removed 39 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**
model_log = ols_object.fit()
model_log.summary()
```

Table 5.9: OLS Regression Results

Dep. Variable:	np.log(price)	R-squared:	0.830
Model:	OLS	Adj. R-squared:	0.829
Method:	Least Squares	F-statistic:	2173.

Date:	Sun, 29 Jan 2023	Prob (F-statistic):	0.00
Time:	01:26:25	Log-Likelihood:	-775.51
No. Observations:	4921	AIC:	1575.
Df Residuals:	4909	BIC:	1653.
Df Model:	11		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-262.7743	24.455	-10.745	0.000	-310.717	-214.832
year	0.1350	0.012	11.148	0.000	0.111	0.159
engineSize	16.6645	5.482	3.040	0.002	5.917	27.412
mileage	0.0008	0.000	5.945	0.000	0.001	0.001
mpg	-1.1217	0.324	-3.458	0.001	-1.758	-0.486
year:engineSize	-0.0081	0.003	-2.997	0.003	-0.013	-0.003
year:mileage	-3.927e-07	6.5e-08	-6.037	0.000	-5.2e-07	-2.65e-07
year:mpg	0.0005	0.000	3.411	0.001	0.000	0.001
engineSize:mileage	-4.566e-07	3.86e-07	-1.183	0.237	-1.21e-06	3e-07
engineSize:mpg	0.0071	0.000	16.202	0.000	0.006	0.008
mileage:mpg	7.29e-08	1.68e-08	4.349	0.000	4e-08	1.06e-07
I(mileage ** 2)	1.418e-11	5.29e-12	2.683	0.007	3.82e-12	2.46e-11

Omnibus:	631.414	Durbin-Watson:	0.553
Prob(Omnibus):	0.000	Jarque-Bera (JB):	1851.015
Skew:	0.682	Prob(JB):	0.00
Kurtosis:	5.677	Cond. No.	1.73e+13

Note that we obtain a higher R-sqauared value of 83% as compared to 80% with the complete data. Removing the influential points helped obtain a 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())
```

```
8820.685844070766
```

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.

5.4 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.

5.4.1 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.**

5.4.2 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:

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

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:

$$VIF(\hat{\beta}_j) = \frac{1}{1 - R^2_{X_j|X_{-j}}} \tag{5.2}$$

```
#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
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?

5.4.3 Manual computation of VIF

```
#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()
```

Table 5.13: OLS Regression Results

Dep. Variable:	year	R-squared:	0.592
Model:	OLS	Adj. R-squared:	0.592
Method:	Least Squares	F-statistic:	2400.
Date:	Mon, 30 Jan 2023	Prob (F-statistic):	0.00
Time:	02:49:19	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

```
#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 5.16: 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 5.19: 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.

5.4.4 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.*

6 Autocorrelation

Read section 3.3.3 (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.

Below is an example showing violation of the autocorrelation assumption *(refer to the book to understand autocorrelation)* in linear regression. Subsequently, it is shown that addressing the assumption violation leads to a much better model fit.

6.1 Introduction

Example: Using linear regression models to predict electricity demand in Toronto, CA.

We have hourly power demand and temperature (in Celsius) data from 2017 to 2020.

We are going to build a linear model to predict the hourly power demand for the next day (for example, when it is 1/1/2021, we predict hourly demand on 1/2/2021 using historical data and the weather forecasts).

When we are building a model, it is important to keep in mind what data we can use as features. For this model:

- We cannot use previous hourly data as features. (Although in a high frequency setting, it is possible)
- The temperature in our raw data can not be used directly, since it is the actual, not the forecasted temperature. We are going to use the previous day temperature as the forecast.

Source: Keep it simple, keep it linear: A linear regression model for time series

```
%pylab inline
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
```

```
plt.rcParams['figure.figsize'] = [9, 5]
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
# A few helper functions
import numpy.ma as ma
from scipy.stats.stats import pearsonr, normaltest
from scipy.spatial.distance import correlation
def build_model(features):
  X=sm.add_constant(df[features])
  y=df['power']
  model = sm.OLS(y,X, missing='drop').fit()
  predictions = model.predict(X)
  display(model.summary())
  res=y-predictions
  return res


def plt_residual(res):
  plt.plot(range(len(res)), res)
  plt.ylabel('Residual')
  plt.xlabel("Hour")

def plt_residual_lag(res, nlag):
  x=res.values
  y=res.shift(nlag).values
  sns.kdeplot(x,y=y,color='blue',shade=True )
  plt.xlabel('res')
  plt.ylabel("res-lag-{}".format(nlag))
  rho,p=corrcoef(x,y)
  plt.title("n_lag={} hours, correlation={:f}".format(nlag, rho))

def plt_acf(res):
  plt.rcParams['figure.figsize'] = [18, 5]
  acorr = sm.tsa.acf(res.dropna(), nlags = len(res.dropna())-1)
  fig, (ax1, ax2) = plt.subplots(1, 2)
  ax1.plot(acorr)
  ax1.set_ylabel('corr')
  ax1.set_xlabel('n_lag')
  ax1.set_title('Auto Correlation')
  ax2.plot(acorr[:4*7*24])
```

```
  ax2.set_ylabel('corr')
  ax2.set_xlabel('n_lag')
  ax2.set_title('Auto Correlation (4-week zoomed in) ')
  plt.show()
  pd.set_option('display.max_columns', None)
  adf=pd.DataFrame(np.round(acorr[:30*24],2).reshape([30, 24] ))
  adf.index.name='day'
  display(adf)
  plt.rcParams['figure.figsize'] = [9, 5]

def corrcoef(x,y):
    a,b=ma.masked_invalid(x),ma.masked_invalid(y)
    msk = (~a.mask & ~b.mask)
    return pearsonr(x[msk],y[msk])[0], normaltest(res, nan_policy='omit')[1]
```

6.2 The data

```
df=pd.read_csv("./Datasets/Toronto_power_demand.csv", parse_dates=['Date'], index_col=0)
df['temperature']=df['temperature'].shift(24*1)
df.tail()
```

key	Date	Hour	power	temperature
20201231:19	2020-12-31	19	5948	4.9
20201231:20	2020-12-31	20	5741	4.5
20201231:21	2020-12-31	21	5527	3.7
20201231:22	2020-12-31	22	5301	2.9
20201231:23	2020-12-31	23	5094	2.1

```
ndays=len(set(df['Date']))
print("There are {} rows, which is {}*24={}, for {} days. And The data is already in sorte
```

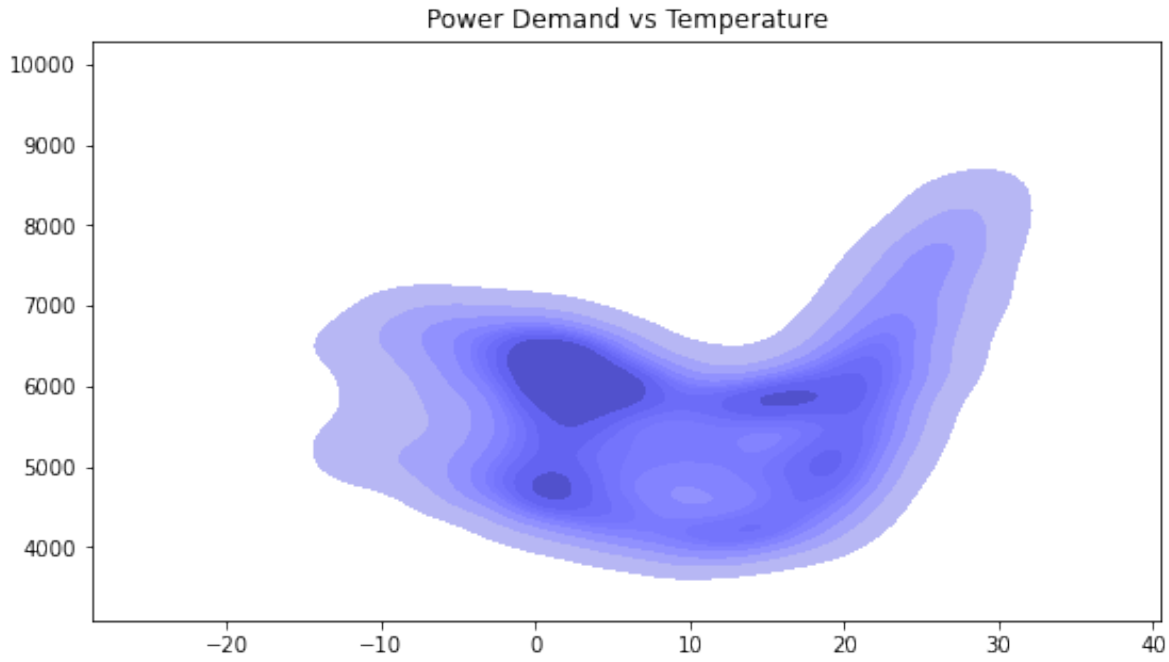
```
There are 35064 rows, which is 1461*24=35064, for 1461 days. And The data is already in sorte
```

```
print("It is natural to think that there is a relationship between power demand and temper
sns.kdeplot(df['temperature'].values, y=df['power'].values,color='blue',shade=True )
```

```
plt.title("Power Demand vs Temperature")
```

It is natural to think that there is a relationship between power demand and temperature.

Text(0.5, 1.0, 'Power Demand vs Temperature')




```
print("""
It is not a linear relationship. We create two features corresponding to hot and cold weat
it possible to develop a linear model.
""")
is_hot=(df['temperature']>15).astype(int)
print("{:f}% of data points are hot".format(is_hot.mean()*100))
df['temp_hot']=df['temperature']*is_hot
df['temp_cold']=df['temperature']*(1-is_hot)
df.tail()
```

It is not a linear relationship. We create two features corresponding to hot and cold weathe

34.813484% of data points are hot

key	Date	Hour	power	temperature	temp_hot	temp_cold
20201231:19	2020-12-31	19	5948	4.9	0.0	4.9
20201231:20	2020-12-31	20	5741	4.5	0.0	4.5
20201231:21	2020-12-31	21	5527	3.7	0.0	3.7
20201231:22	2020-12-31	22	5301	2.9	0.0	2.9
20201231:23	2020-12-31	23	5094	2.1	0.0	2.1

6.3 Predictor: temperature

```
res=build_model(['temp_hot', 'temp_cold'])
```

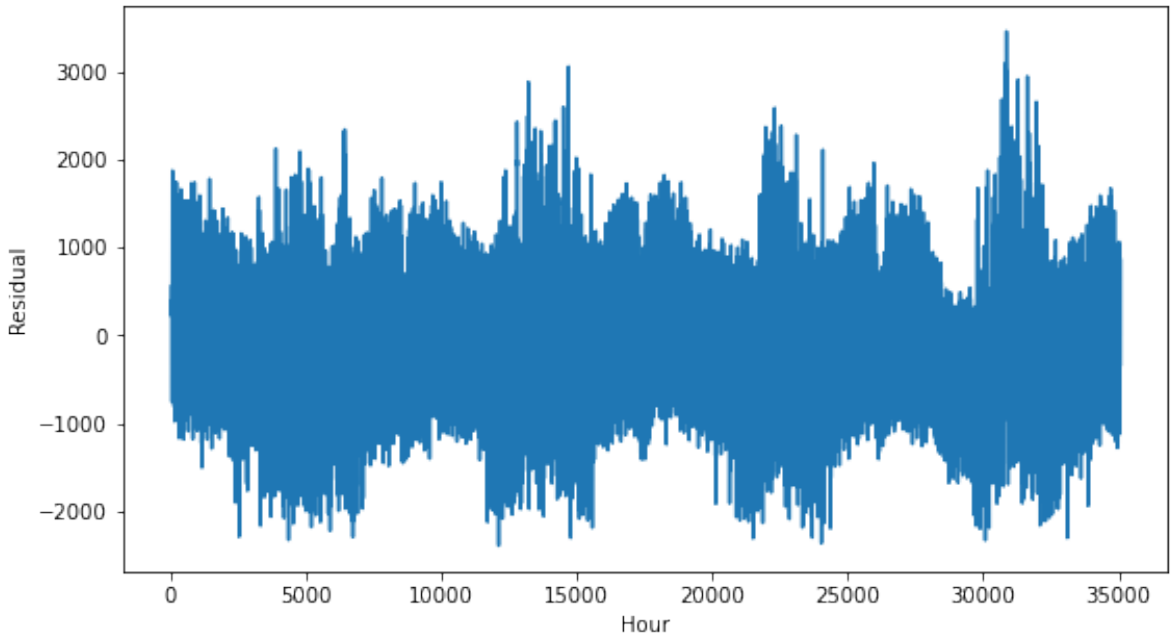
Table 6.3: OLS Regression Results

Dep. Variable:	power	R-squared:	0.195
Model:	OLS	Adj. R-squared:	0.195
Method:	Least Squares	F-statistic:	4251.
Date:	Sun, 05 Feb 2023	Prob (F-statistic):	0.00
Time:	23:15:53	Log-Likelihood:	-2.8766e+05
No. Observations:	35040	AIC:	5.753e+05
Df Residuals:	35037	BIC:	5.753e+05
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	5501.3027	6.222	884.115	0.000	5489.107	5513.499
temp_hot	31.8488	0.462	68.911	0.000	30.943	32.755
temp_cold	-37.5088	0.827	-45.364	0.000	-39.129	-35.888

Omnibus:	945.032	Durbin-Watson:	0.093
Prob(Omnibus):	0.000	Jarque-Bera (JB):	469.200
Skew:	0.034	Prob(JB):	1.30e-102
Kurtosis:	2.437	Cond. No.	17.0

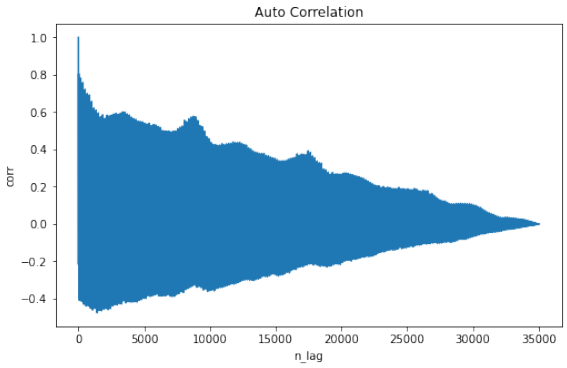
```
plt_residual(res)
```

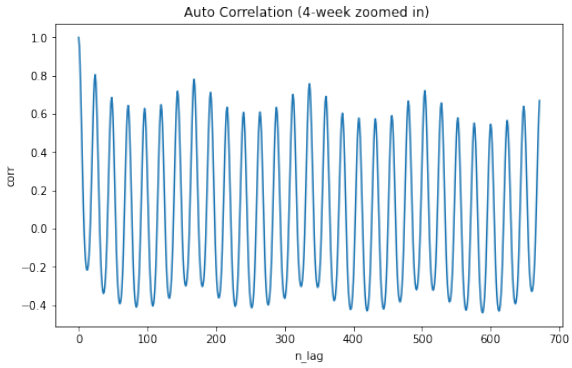



```
print("acf shows that there is a strong correlation for 24 lags, which is one day.")
plt_acf(res)
```

```
acf shows that there is a strong correlation for 24 lags, which is one day.
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\statsmodels\tsa\stattools.py:667: FutureWarning
  warnings.warn(
```

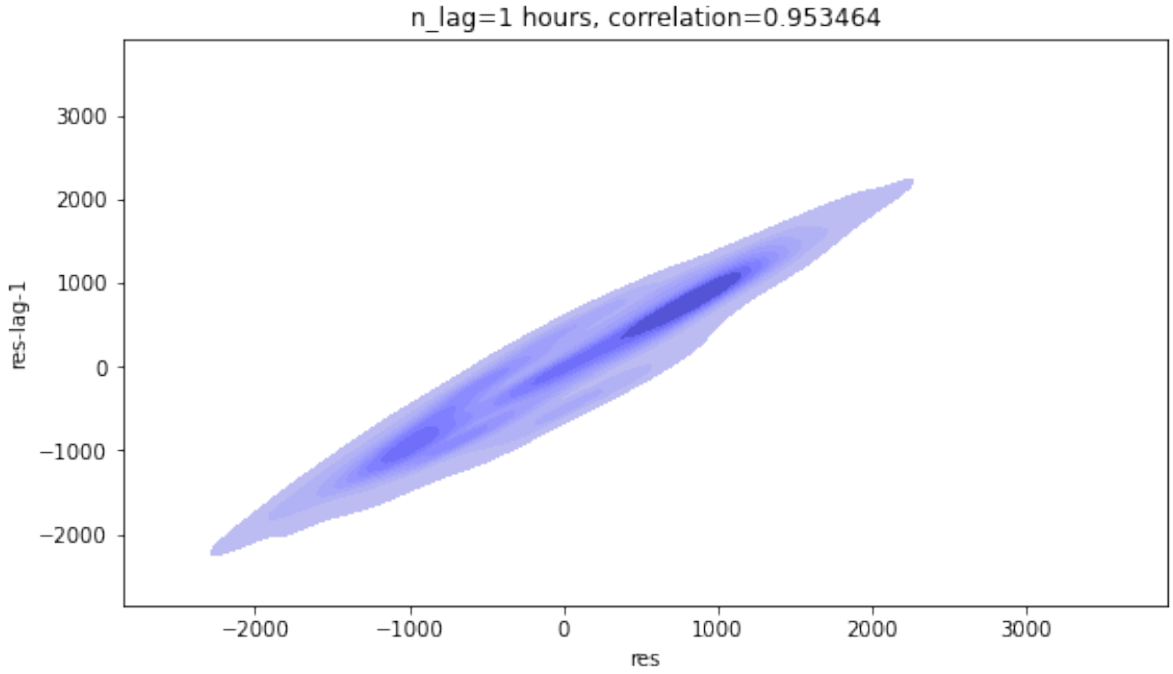


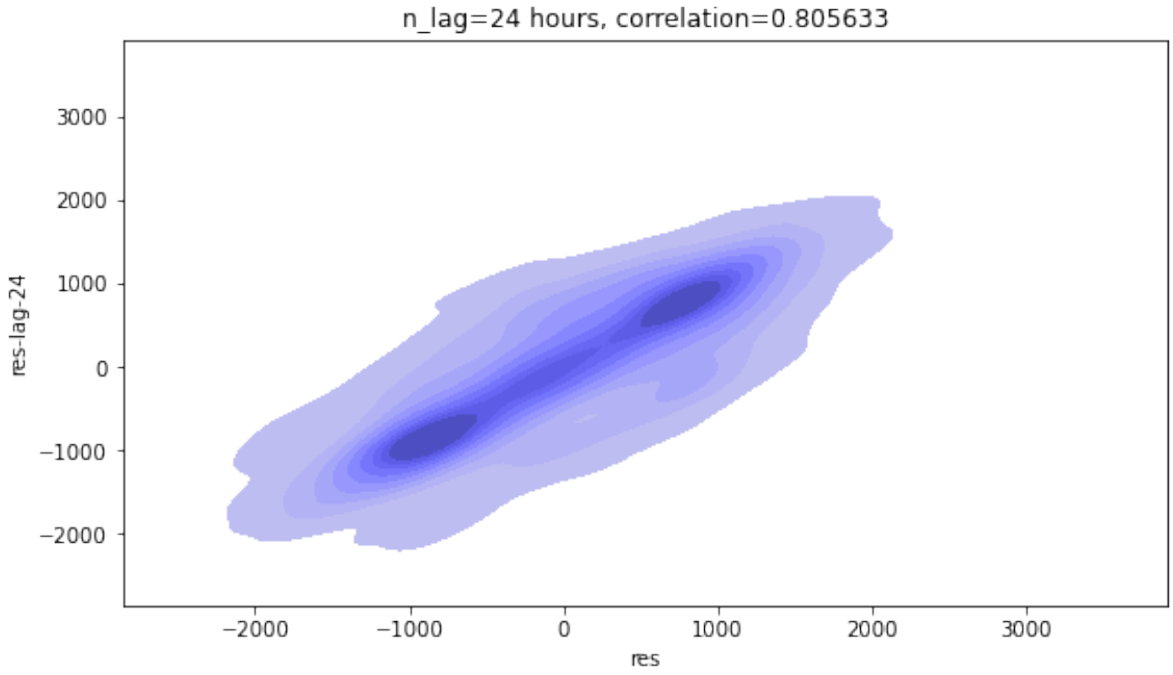


day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.00	0.95	0.85	0.72	0.56	0.40	0.24	0.09	-0.02	-0.11	-0.16	-0.20	-0.22	-0.21	-0.19
1	0.81	0.77	0.68	0.55	0.40	0.25	0.09	-0.04	-0.15	-0.23	-0.29	-0.32	-0.34	-0.33	-0.31
2	0.69	0.65	0.57	0.45	0.31	0.16	0.01	-0.12	-0.22	-0.30	-0.35	-0.38	-0.39	-0.38	-0.36
3	0.64	0.61	0.53	0.42	0.28	0.13	-0.01	-0.14	-0.24	-0.32	-0.37	-0.40	-0.41	-0.40	-0.37
4	0.63	0.60	0.52	0.41	0.27	0.12	-0.02	-0.14	-0.24	-0.32	-0.37	-0.40	-0.40	-0.39	-0.36
5	0.65	0.62	0.54	0.43	0.30	0.15	0.01	-0.11	-0.21	-0.28	-0.33	-0.36	-0.36	-0.35	-0.32
6	0.72	0.69	0.61	0.50	0.36	0.21	0.07	-0.05	-0.15	-0.22	-0.27	-0.29	-0.30	-0.29	-0.26
7	0.78	0.75	0.67	0.54	0.40	0.25	0.10	-0.03	-0.13	-0.21	-0.26	-0.29	-0.30	-0.30	-0.27
8	0.71	0.68	0.60	0.48	0.34	0.19	0.04	-0.09	-0.19	-0.27	-0.32	-0.35	-0.36	-0.36	-0.33
9	0.64	0.61	0.53	0.41	0.27	0.13	-0.02	-0.14	-0.24	-0.32	-0.37	-0.40	-0.41	-0.40	-0.37
10	0.61	0.58	0.50	0.39	0.25	0.11	-0.03	-0.16	-0.26	-0.33	-0.38	-0.40	-0.41	-0.40	-0.38
11	0.61	0.58	0.51	0.39	0.26	0.12	-0.02	-0.14	-0.24	-0.32	-0.36	-0.39	-0.40	-0.39	-0.36
12	0.63	0.61	0.53	0.42	0.28	0.14	0.00	-0.12	-0.22	-0.29	-0.33	-0.36	-0.36	-0.35	-0.32
13	0.70	0.67	0.60	0.48	0.35	0.20	0.06	-0.06	-0.16	-0.23	-0.27	-0.30	-0.30	-0.29	-0.26
14	0.76	0.73	0.64	0.52	0.38	0.23	0.09	-0.04	-0.14	-0.22	-0.27	-0.30	-0.31	-0.30	-0.27
15	0.69	0.66	0.58	0.46	0.32	0.17	0.03	-0.10	-0.20	-0.28	-0.33	-0.36	-0.38	-0.37	-0.35
16	0.60	0.57	0.50	0.38	0.25	0.10	-0.04	-0.16	-0.26	-0.34	-0.38	-0.41	-0.42	-0.41	-0.39
17	0.58	0.55	0.47	0.36	0.23	0.09	-0.05	-0.17	-0.27	-0.34	-0.39	-0.42	-0.43	-0.42	-0.39
18	0.57	0.55	0.47	0.36	0.23	0.09	-0.05	-0.17	-0.27	-0.34	-0.39	-0.41	-0.42	-0.41	-0.38
19	0.59	0.57	0.49	0.38	0.25	0.11	-0.03	-0.14	-0.24	-0.31	-0.35	-0.38	-0.38	-0.37	-0.34
20	0.67	0.64	0.56	0.45	0.32	0.18	0.04	-0.08	-0.17	-0.24	-0.29	-0.31	-0.32	-0.31	-0.28
21	0.72	0.69	0.61	0.49	0.36	0.21	0.07	-0.06	-0.16	-0.23	-0.28	-0.31	-0.32	-0.32	-0.29
22	0.66	0.63	0.55	0.43	0.29	0.15	0.01	-0.12	-0.22	-0.29	-0.34	-0.37	-0.38	-0.38	-0.35
23	0.58	0.55	0.47	0.36	0.23	0.09	-0.05	-0.17	-0.27	-0.34	-0.39	-0.42	-0.43	-0.42	-0.39
24	0.55	0.52	0.45	0.34	0.21	0.07	-0.07	-0.19	-0.29	-0.36	-0.40	-0.43	-0.44	-0.43	-0.40
25	0.55	0.52	0.45	0.34	0.21	0.07	-0.07	-0.19	-0.28	-0.35	-0.40	-0.42	-0.43	-0.42	-0.39
26	0.57	0.54	0.47	0.36	0.23	0.09	-0.04	-0.16	-0.25	-0.32	-0.36	-0.39	-0.39	-0.38	-0.35
27	0.64	0.61	0.54	0.43	0.30	0.16	0.03	-0.09	-0.19	-0.25	-0.30	-0.32	-0.33	-0.32	-0.29
28	0.70	0.67	0.59	0.47	0.34	0.19	0.05	-0.07	-0.17	-0.24	-0.29	-0.32	-0.33	-0.33	-0.30
29	0.63	0.60	0.53	0.41	0.28	0.13	-0.01	-0.13	-0.23	-0.30	-0.35	-0.38	-0.39	-0.39	-0.37

```
print("Although 1 hour lag correlation is more strong, but we cannot use it, as we intend
the power consumption for the next day.")
plt_residual_lag(res,1)
plt.show()
plt_residual_lag(res,24)
```

```
Although 1 hour lag correlation is more strong, but we cannot use it, as we intend to predict
```


n_lag=1 hours, correlation=0.953464
res-lag-1
res
3000
2000
1000
0
−1000
−2000
−2000
−1000
0
1000
2000
3000


n_lag=24 hours, correlation=0.805633
res-lag-24
res
3000
2000
1000
0
−1000
−2000
−2000
−1000
0
1000
2000
3000

6.4 Predictors: Temperature + one day lag of power.

```
df['power_lag_1_day']=df['power'].shift(24)
df.tail()
```

	key	Date	Hour	power	temperature	temp_hot	temp_cold	power_lag_1_day
35059	20201231:19	2020-12-31	19	5948	4.9	0.0	4.9	6163.0
35060	20201231:20	2020-12-31	20	5741	4.5	0.0	4.5	5983.0
35061	20201231:21	2020-12-31	21	5527	3.7	0.0	3.7	5727.0
35062	20201231:22	2020-12-31	22	5301	2.9	0.0	2.9	5428.0
35063	20201231:23	2020-12-31	23	5094	2.1	0.0	2.1	5104.0

```
res=build_model(['temp_hot', 'temp_cold', 'power_lag_1_day' ])
```

```
/usr/local/lib/python3.8/dist-packages/statsmodels/tsa/tsatools.py:142: FutureWarning: In a f
  x = pd.concat(x[::order], 1)
```

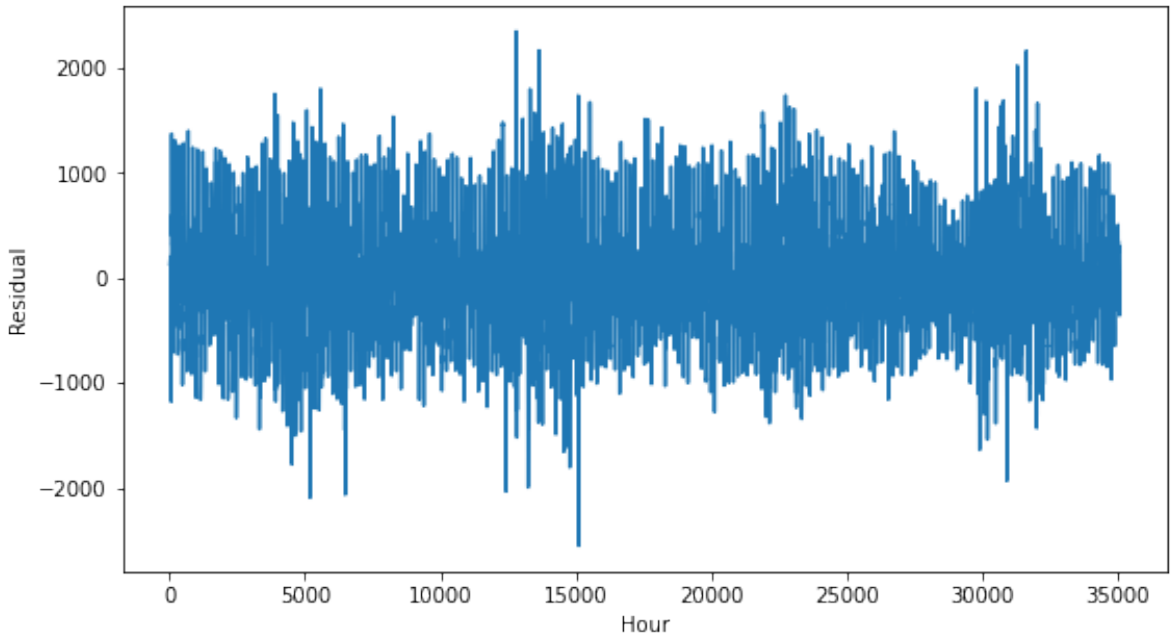
Table 6.8: OLS Regression Results

Dep. Variable:	power	R-squared:	0.794
Model:	OLS	Adj. R-squared:	0.794
Method:	Least Squares	F-statistic:	4.513e+04
Date:	Sun, 22 Jan 2023	Prob (F-statistic):	0.00
Time:	19:21:14	Log-Likelihood:	-2.6375e+05
No. Observations:	35040	AIC:	5.275e+05
Df Residuals:	35036	BIC:	5.275e+05
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	689.2701	15.384	44.806	0.000	659.118	719.422
temp_hot	3.2158	0.250	12.853	0.000	2.725	3.706
temp_cold	-1.3464	0.433	-3.110	0.002	-2.195	-0.498
power_lag_1_day	0.8747	0.003	319.552	0.000	0.869	0.880

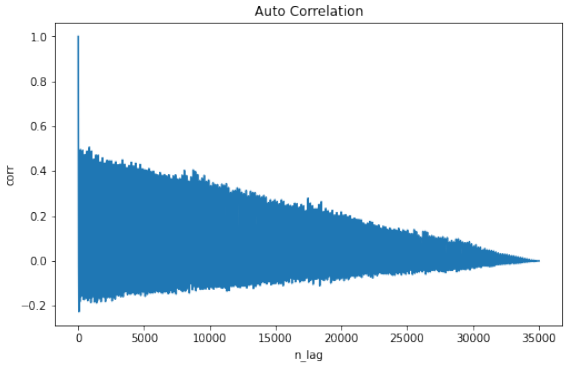
Omnibus:	2035.537	Durbin-Watson:	0.041
Prob(Omnibus):	0.000	Jarque-Bera (JB):	5794.290
Skew:	0.301	Prob(JB):	0.00
Kurtosis:	4.899	Cond. No.	3.69e+04

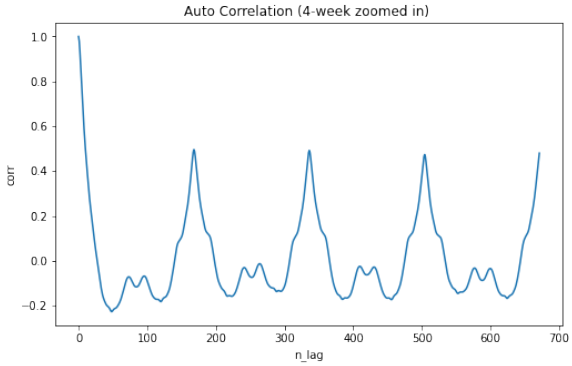
```
plt_residual(res)
```




```
plt_acf(res)
```

```
/usr/local/lib/python3.8/dist-packages/statsmodels/tsa/stattools.py:667: FutureWarning: fft=T
  warnings.warn(
```

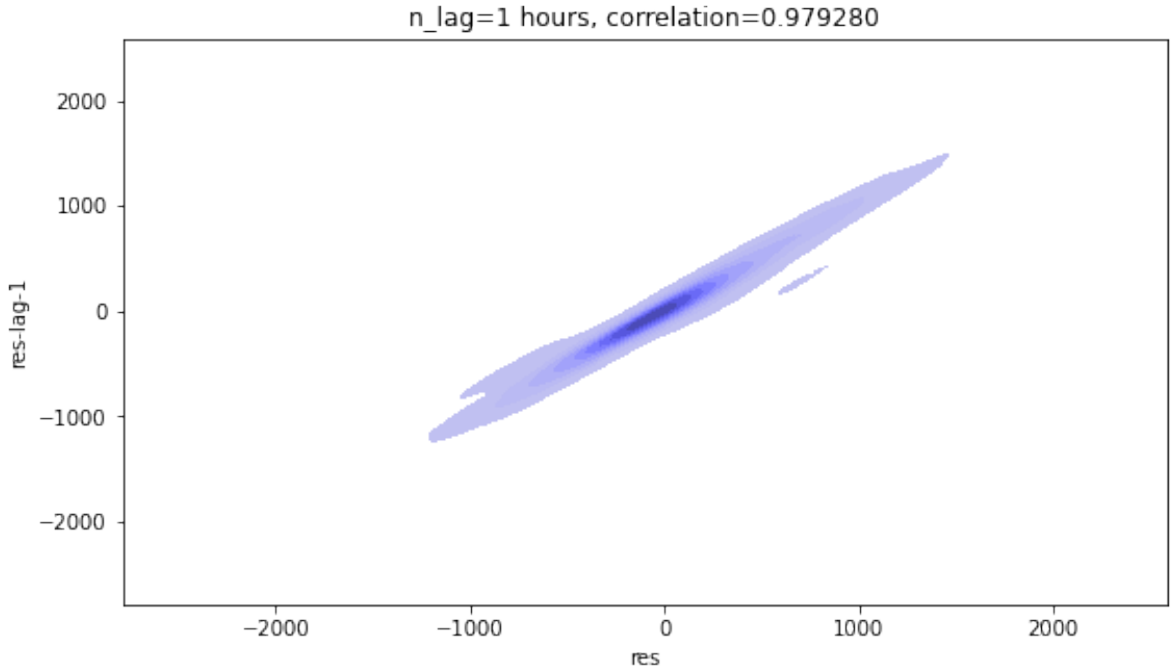




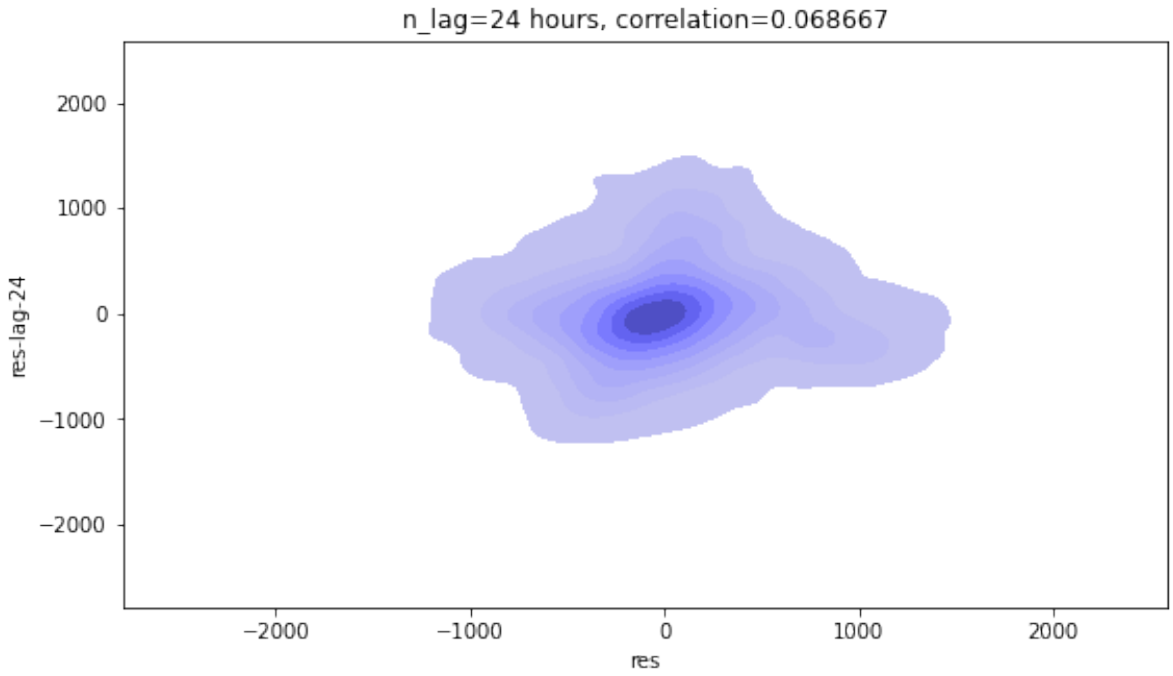
day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.00	0.98	0.93	0.87	0.81	0.75	0.70	0.64	0.59	0.54	0.50	0.46	0.42	0.39	0.3
1	0.07	0.05	0.03	0.00	-0.02	-0.04	-0.06	-0.08	-0.10	-0.12	-0.14	-0.15	-0.16	-0.17	-0
2	-0.23	-0.23	-0.22	-0.22	-0.21	-0.21	-0.21	-0.21	-0.21	-0.20	-0.20	-0.19	-0.18	-0.18	-0
3	-0.07	-0.07	-0.07	-0.07	-0.08	-0.09	-0.09	-0.10	-0.11	-0.11	-0.11	-0.12	-0.12	-0.12	-0
4	-0.07	-0.07	-0.07	-0.08	-0.09	-0.10	-0.11	-0.12	-0.13	-0.14	-0.14	-0.15	-0.16	-0.16	-0
5	-0.18	-0.18	-0.17	-0.17	-0.17	-0.16	-0.16	-0.16	-0.16	-0.15	-0.14	-0.14	-0.13	-0.12	-0
6	0.07	0.08	0.09	0.09	0.10	0.10	0.11	0.12	0.13	0.14	0.16	0.18	0.19	0.21	0.2
7	0.50	0.49	0.46	0.43	0.40	0.37	0.34	0.31	0.28	0.26	0.24	0.22	0.21	0.19	0.1
8	0.10	0.09	0.07	0.06	0.04	0.02	-0.00	-0.02	-0.04	-0.05	-0.07	-0.08	-0.09	-0.10	-0
9	-0.16	-0.16	-0.16	-0.15	-0.16	-0.16	-0.16	-0.16	-0.16	-0.16	-0.15	-0.15	-0.14	-0.14	-0
10	-0.03	-0.03	-0.03	-0.03	-0.04	-0.04	-0.05	-0.06	-0.06	-0.07	-0.07	-0.07	-0.07	-0.07	-0
11	-0.01	-0.01	-0.02	-0.03	-0.03	-0.04	-0.05	-0.06	-0.07	-0.08	-0.09	-0.10	-0.11	-0.11	-0
12	-0.14	-0.14	-0.13	-0.13	-0.13	-0.13	-0.14	-0.14	-0.13	-0.13	-0.13	-0.12	-0.11	-0.10	-0
13	0.08	0.09	0.10	0.10	0.11	0.11	0.12	0.13	0.14	0.15	0.17	0.18	0.20	0.22	0.2
14	0.49	0.48	0.46	0.43	0.40	0.37	0.34	0.31	0.28	0.26	0.24	0.23	0.21	0.20	0.1
15	0.10	0.09	0.07	0.05	0.03	0.01	-0.01	-0.03	-0.05	-0.07	-0.08	-0.10	-0.11	-0.12	-0
16	-0.17	-0.17	-0.17	-0.17	-0.16	-0.16	-0.17	-0.17	-0.16	-0.16	-0.16	-0.16	-0.15	-0.14	-0
17	-0.03	-0.02	-0.02	-0.03	-0.03	-0.04	-0.04	-0.05	-0.05	-0.06	-0.06	-0.06	-0.06	-0.06	-0
18	-0.03	-0.04	-0.04	-0.05	-0.06	-0.08	-0.09	-0.10	-0.11	-0.12	-0.13	-0.14	-0.15	-0.15	-0
19	-0.17	-0.17	-0.16	-0.16	-0.15	-0.15	-0.15	-0.15	-0.14	-0.13	-0.13	-0.12	-0.11	-0.10	-0
20	0.10	0.10	0.11	0.11	0.12	0.12	0.12	0.13	0.14	0.15	0.17	0.18	0.20	0.21	0.2
21	0.47	0.46	0.44	0.41	0.38	0.35	0.32	0.29	0.26	0.24	0.22	0.21	0.19	0.18	0.1
22	0.10	0.09	0.07	0.05	0.03	0.01	-0.00	-0.02	-0.04	-0.05	-0.07	-0.08	-0.09	-0.10	-0
23	-0.14	-0.14	-0.14	-0.14	-0.14	-0.14	-0.14	-0.14	-0.14	-0.14	-0.13	-0.13	-0.13	-0.12	-0
24	-0.03	-0.03	-0.03	-0.04	-0.05	-0.05	-0.06	-0.07	-0.08	-0.08	-0.09	-0.09	-0.09	-0.09	-0
25	-0.03	-0.04	-0.04	-0.05	-0.06	-0.07	-0.08	-0.09	-0.10	-0.11	-0.12	-0.13	-0.14	-0.14	-0
26	-0.17	-0.17	-0.16	-0.16	-0.16	-0.15	-0.15	-0.15	-0.15	-0.14	-0.13	-0.13	-0.12	-0.11	-0
27	0.08	0.09	0.10	0.11	0.11	0.12	0.12	0.13	0.14	0.16	0.18	0.19	0.21	0.22	0.2

day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
28	0.49	0.48	0.45	0.42	0.39	0.36	0.33	0.30	0.27	0.25	0.23	0.22	0.20	0.19	0.
29	0.09	0.08	0.06	0.04	0.02	-0.00	-0.02	-0.04	-0.06	-0.08	-0.09	-0.10	-0.12	-0.13	-0

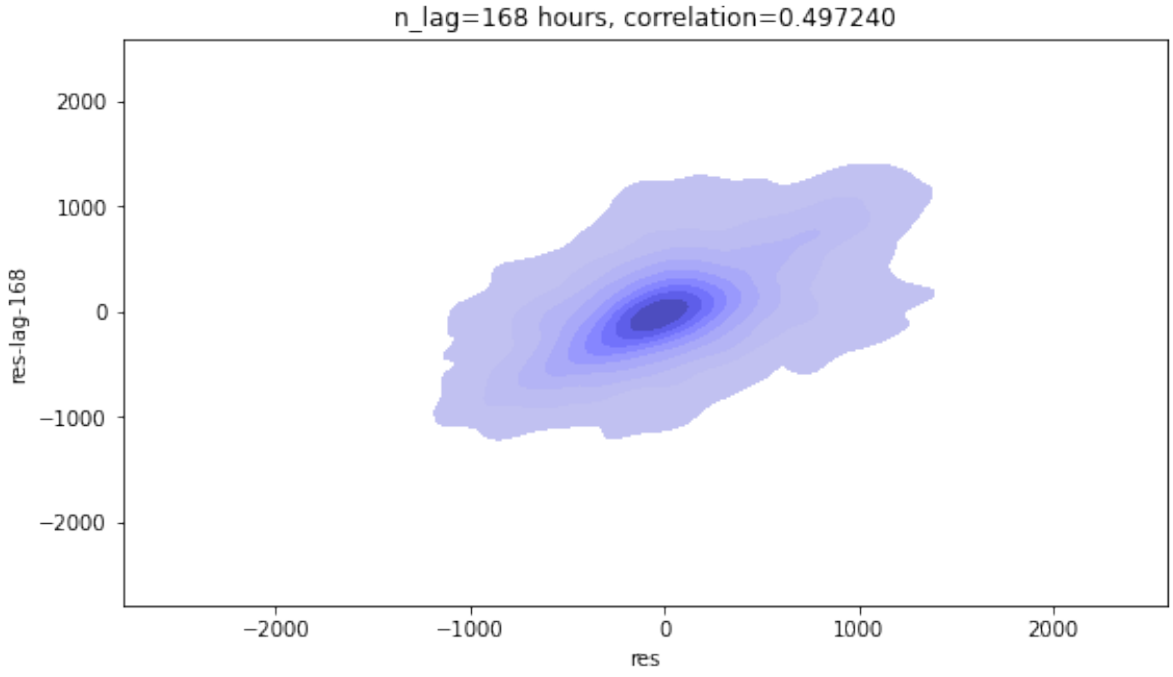
```
plt_residual_lag(res, 1)
```




```
plt_residual_lag(res, 24)
```




```
plt_residual_lag(res, 24*7)
```



6.5 Predictors: Temperature + 1 day lag of power + 1 week lag of power

```
df['power_lag_1_week']=df['power'].shift(24*7)
df.tail()
```

	key	Date	Hour	power	temperature	temp_hot	temp_cold	power_lag_1_day
35059	20201231:19	2020-12-31	19	5948	4.9	0.0	4.9	6163.0
35060	20201231:20	2020-12-31	20	5741	4.5	0.0	4.5	5983.0
35061	20201231:21	2020-12-31	21	5527	3.7	0.0	3.7	5727.0
35062	20201231:22	2020-12-31	22	5301	2.9	0.0	2.9	5428.0
35063	20201231:23	2020-12-31	23	5094	2.1	0.0	2.1	5104.0

```
res=build_model(['temp_hot', 'temp_cold', 'power_lag_1_day', 'power_lag_1_week' ])
```

```
/usr/local/lib/python3.8/dist-packages/statsmodels/tsa/tsatools.py:142: FutureWarning: In a f
  x = pd.concat(x[::order], 1)
```

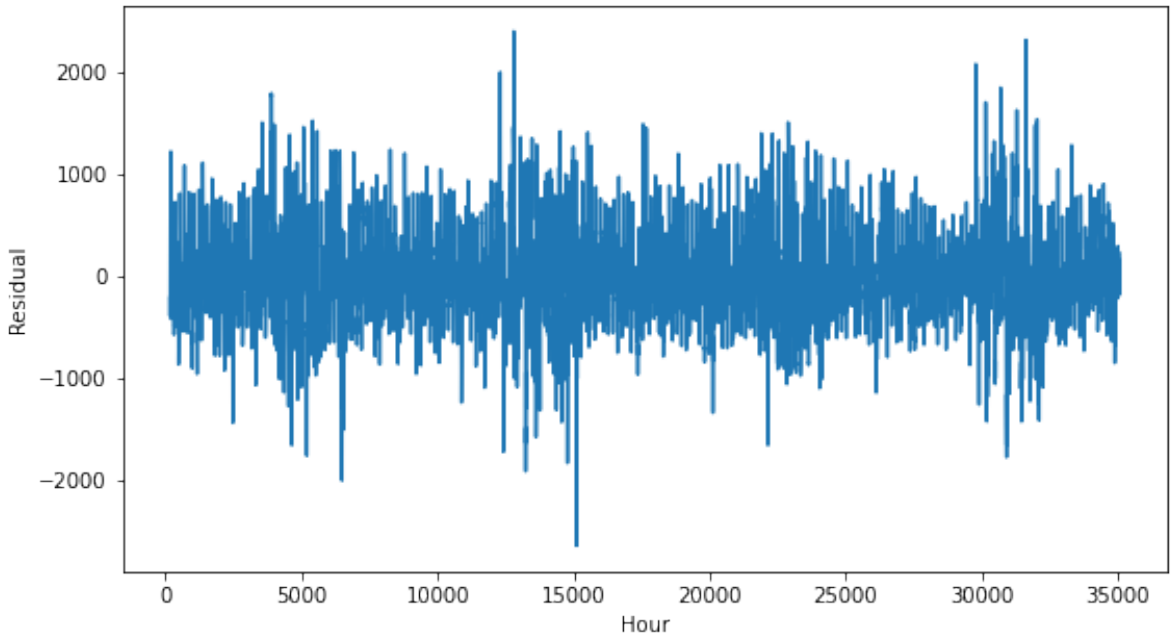
Table 6.13: OLS Regression Results

Dep. Variable:	power	R-squared:	0.840
Model:	OLS	Adj. R-squared:	0.840
Method:	Least Squares	F-statistic:	4.585e+04
Date:	Sun, 22 Jan 2023	Prob (F-statistic):	0.00
Time:	19:22:49	Log-Likelihood:	-2.5830e+05
No. Observations:	34896	AIC:	5.166e+05
Df Residuals:	34891	BIC:	5.167e+05
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	290.4344	14.166	20.502	0.000	262.668	318.201
temp_hot	3.2967	0.221	14.896	0.000	2.863	3.730
temp_cold	-4.5938	0.385	-11.943	0.000	-5.348	-3.840
power_lag_1_day	0.6114	0.004	170.709	0.000	0.604	0.618
power_lag_1_week	0.3342	0.003	99.595	0.000	0.328	0.341

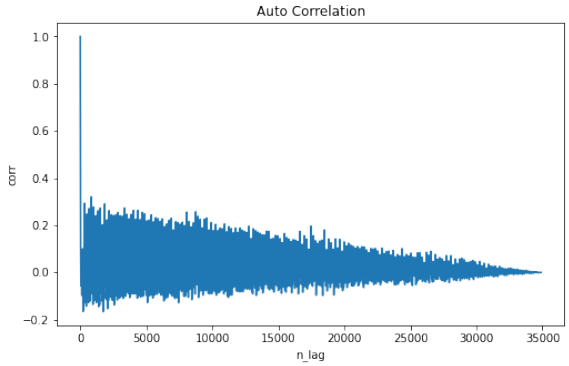
Omnibus:	2729.372	Durbin-Watson:	0.037
Prob(Omnibus):	0.000	Jarque-Bera (JB):	11234.560
Skew:	0.299	Prob(JB):	0.00
Kurtosis:	5.715	Cond. No.	5.43e+04

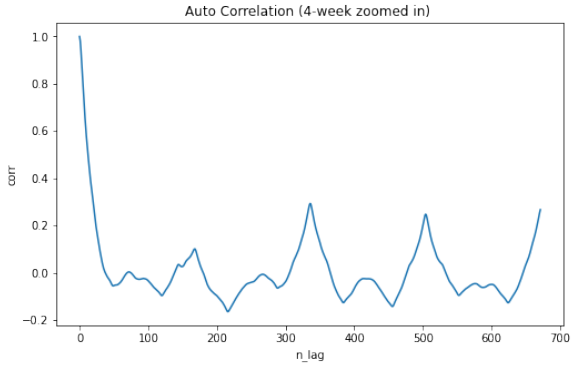
```
plt_residual(res)
```




```
plt_acf(res)
```

```
/usr/local/lib/python3.8/dist-packages/statsmodels/tsa/stattools.py:667: FutureWarning: fft=T
  warnings.warn(
```

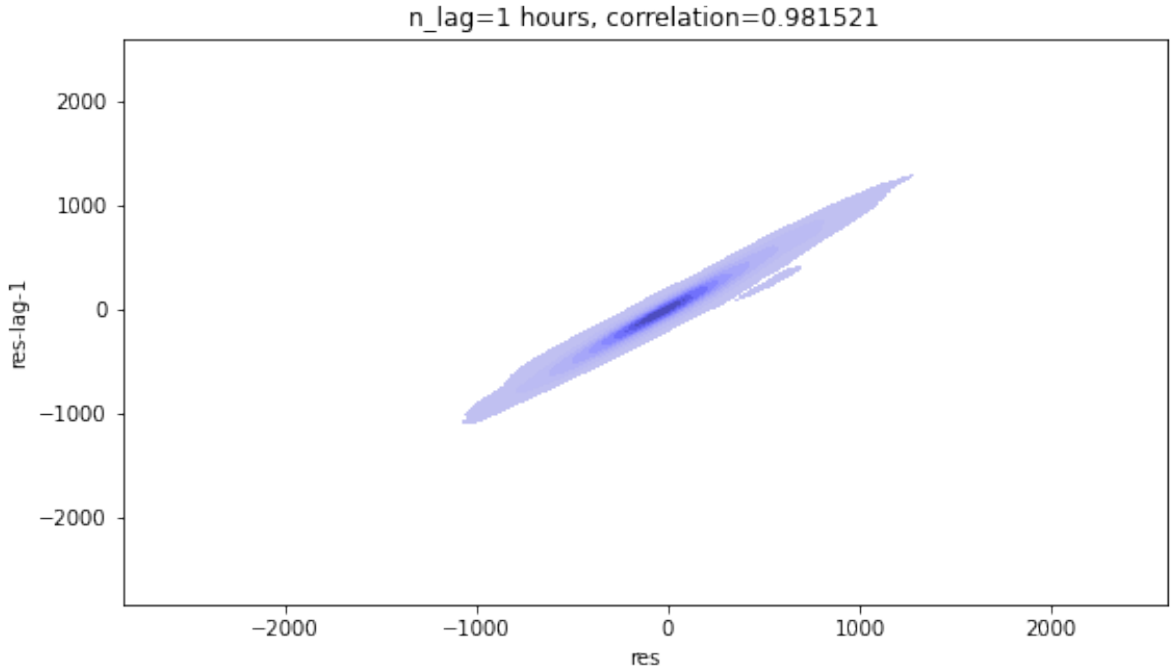




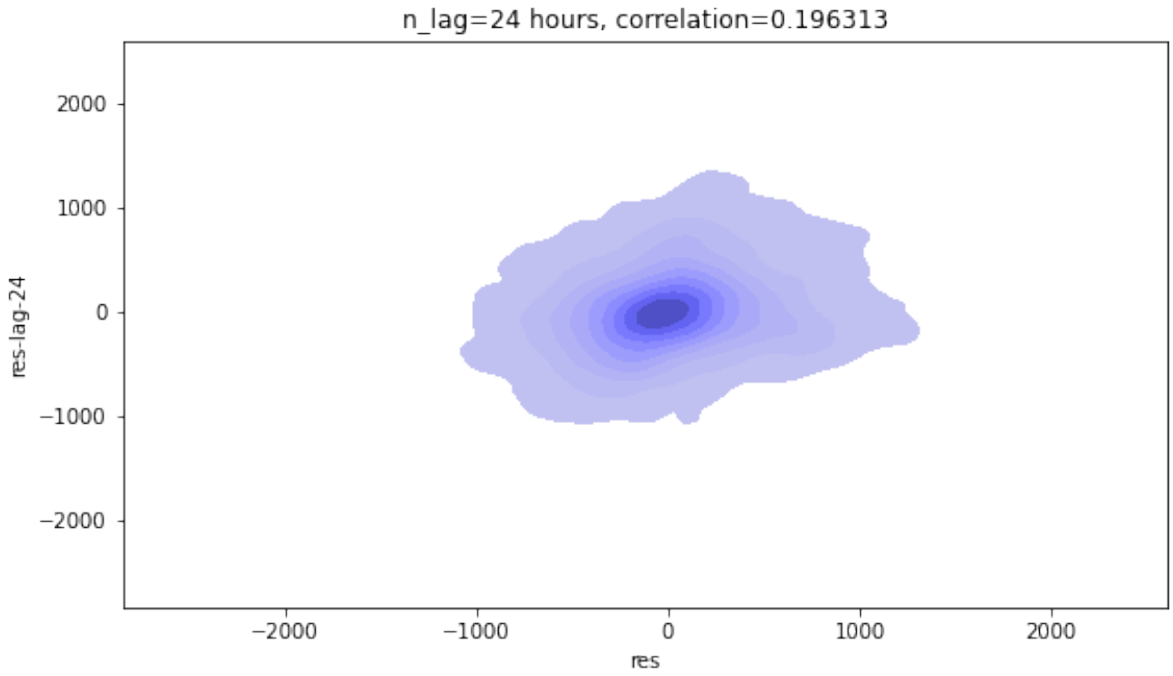
day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.00	0.98	0.94	0.89	0.84	0.79	0.74	0.70	0.65	0.61	0.58	0.54	0.51	0.48	0.
1	0.20	0.18	0.16	0.14	0.12	0.10	0.09	0.07	0.06	0.04	0.03	0.02	0.01	0.00	-0
2	-0.05	-0.06	-0.06	-0.05	-0.05	-0.05	-0.05	-0.05	-0.05	-0.05	-0.05	-0.04	-0.04	-0.04	-0
3	0.00	0.00	0.00	-0.00	-0.00	-0.01	-0.01	-0.01	-0.02	-0.02	-0.02	-0.03	-0.03	-0.03	-0
4	-0.03	-0.03	-0.03	-0.03	-0.03	-0.04	-0.04	-0.04	-0.05	-0.05	-0.05	-0.06	-0.06	-0.06	-0
5	-0.10	-0.09	-0.09	-0.08	-0.08	-0.07	-0.07	-0.07	-0.06	-0.06	-0.05	-0.05	-0.04	-0.04	-0
6	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.04	0.04	0.05	0.05	0.05	0.
7	0.10	0.10	0.09	0.08	0.06	0.05	0.05	0.04	0.03	0.02	0.01	0.01	0.00	-0.01	-0
8	-0.08	-0.08	-0.08	-0.09	-0.09	-0.09	-0.09	-0.10	-0.10	-0.10	-0.10	-0.11	-0.11	-0.11	-0
9	-0.17	-0.16	-0.16	-0.15	-0.15	-0.14	-0.14	-0.13	-0.13	-0.12	-0.12	-0.11	-0.11	-0.10	-0
10	-0.06	-0.05	-0.05	-0.05	-0.05	-0.05	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04	-0
11	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.02	-0.02	-0.02	-0.03	-0.03	-0
12	-0.07	-0.07	-0.06	-0.06	-0.06	-0.06	-0.06	-0.05	-0.05	-0.05	-0.05	-0.04	-0.04	-0.03	-0
13	0.05	0.06	0.06	0.07	0.08	0.08	0.09	0.10	0.11	0.12	0.13	0.14	0.15	0.16	0.
14	0.29	0.29	0.27	0.26	0.24	0.23	0.21	0.20	0.19	0.18	0.17	0.16	0.15	0.14	0.
15	0.05	0.04	0.03	0.02	0.01	-0.00	-0.01	-0.02	-0.03	-0.04	-0.05	-0.06	-0.06	-0.07	-0
16	-0.13	-0.13	-0.12	-0.12	-0.11	-0.11	-0.11	-0.10	-0.10	-0.10	-0.10	-0.09	-0.09	-0.08	-0
17	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0
18	-0.05	-0.05	-0.06	-0.06	-0.06	-0.07	-0.07	-0.08	-0.08	-0.09	-0.09	-0.10	-0.10	-0.10	-0
19	-0.14	-0.14	-0.13	-0.13	-0.12	-0.11	-0.11	-0.10	-0.10	-0.09	-0.08	-0.08	-0.07	-0.06	-0
20	0.03	0.03	0.04	0.04	0.05	0.05	0.06	0.07	0.07	0.08	0.09	0.10	0.11	0.12	0.
21	0.25	0.24	0.23	0.22	0.20	0.19	0.17	0.16	0.14	0.13	0.12	0.11	0.11	0.10	0.
22	0.04	0.03	0.02	0.02	0.01	0.00	-0.01	-0.01	-0.02	-0.03	-0.03	-0.04	-0.04	-0.05	-0
23	-0.10	-0.10	-0.09	-0.09	-0.09	-0.08	-0.08	-0.08	-0.07	-0.07	-0.07	-0.07	-0.07	-0.06	-0
24	-0.05	-0.05	-0.05	-0.05	-0.05	-0.05	-0.06	-0.06	-0.06	-0.06	-0.06	-0.06	-0.06	-0.06	-0
25	-0.05	-0.05	-0.05	-0.05	-0.05	-0.06	-0.06	-0.06	-0.07	-0.07	-0.08	-0.08	-0.09	-0.09	-0
26	-0.13	-0.13	-0.12	-0.12	-0.11	-0.11	-0.10	-0.10	-0.09	-0.09	-0.08	-0.08	-0.07	-0.07	-0
27	0.02	0.03	0.03	0.04	0.05	0.06	0.06	0.07	0.08	0.10	0.11	0.12	0.13	0.14	0.

day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
28	0.27	0.27	0.25	0.24	0.22	0.21	0.19	0.18	0.17	0.15	0.14	0.14	0.13	0.12	0.
29	0.04	0.03	0.02	0.02	0.01	-0.00	-0.01	-0.02	-0.03	-0.04	-0.04	-0.05	-0.06	-0.07	-0

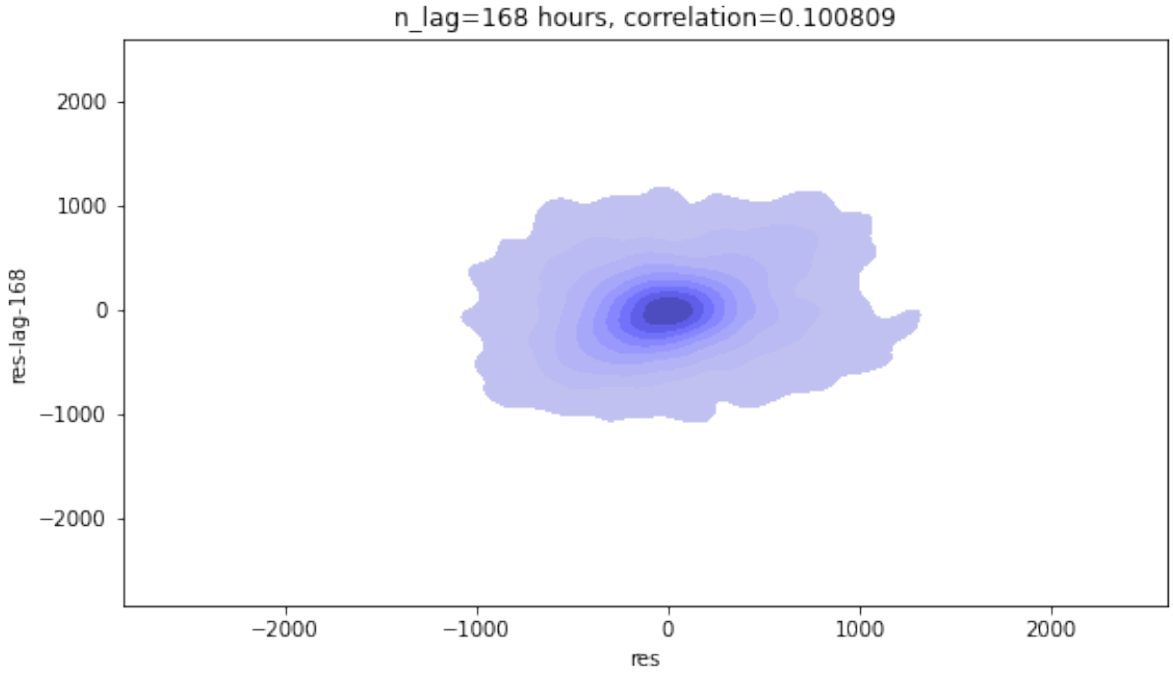
```
plt_residual_lag(res, 1)
```



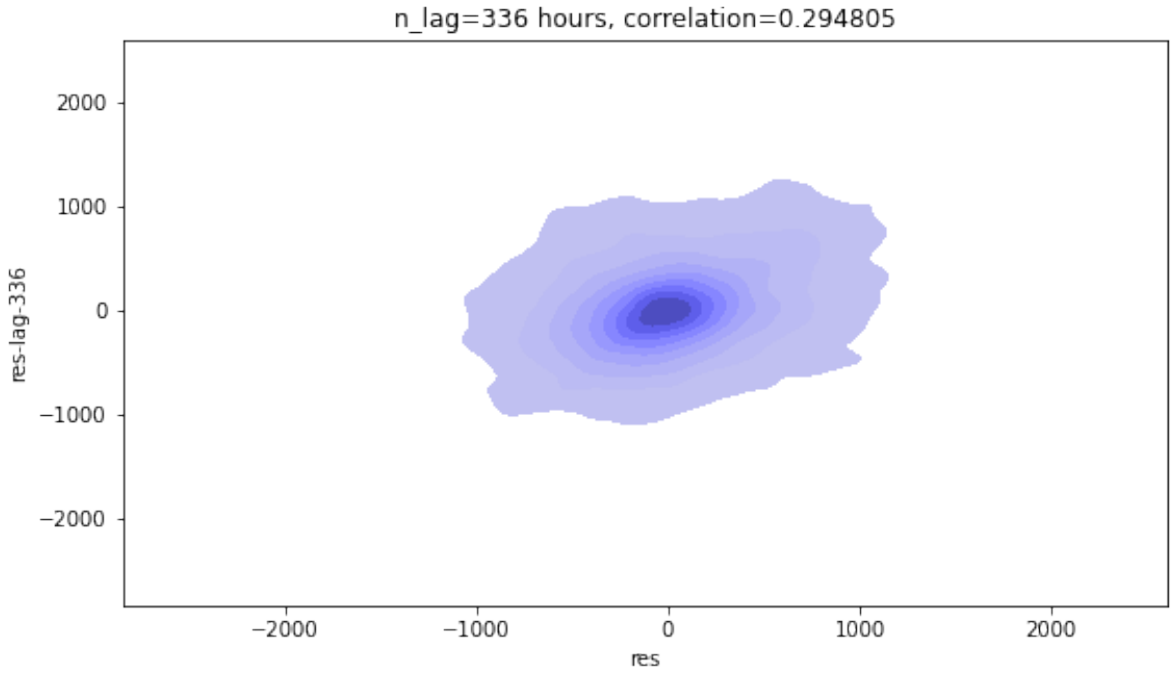

```
plt_residual_lag(res, 24)
```




```
plt_residual_lag(res, 24*7)
```



```
plt_residual_lag(res, 24*7*2)
```



6.6 Predictors: Temperature + 1 day lag of power + 1 week lag of power + 2 weeks lag of power

Although the data shows there is a significant (but not strong) correlation, we need to be cautious to use this feature because there are no simple reasons for this relationship.

For 1-day-lag feature, the correlation is easily understood.

For 1-week-lag feature, we could argue that the behaviour is different between weekday and weekend.

But for 2-week-lag feature, it is hard to understand especially when we have included 1-day-lag and 1-week-lag features. The relation is spurious.

```
df['power_lag_2_week']=df['power'].shift(24*7*2)
df.tail()
```

	key	Date	Hour	power	temperature	temp_hot	temp_cold	power_lag_1_day
35059	20201231:19	2020-12-31	19	5948	4.9	0.0	4.9	6163.0
35060	20201231:20	2020-12-31	20	5741	4.5	0.0	4.5	5983.0
35061	20201231:21	2020-12-31	21	5527	3.7	0.0	3.7	5727.0
35062	20201231:22	2020-12-31	22	5301	2.9	0.0	2.9	5428.0
35063	20201231:23	2020-12-31	23	5094	2.1	0.0	2.1	5104.0

```
res=build_model(['temp_hot', 'temp_cold', 'power_lag_1_day','power_lag_1_week', 'power_lag
```

```
/usr/local/lib/python3.8/dist-packages/statsmodels/tsa/tsatools.py:142: FutureWarning: In a f
  x = pd.concat(x[::order], 1)
```

Table 6.18: OLS Regression Results

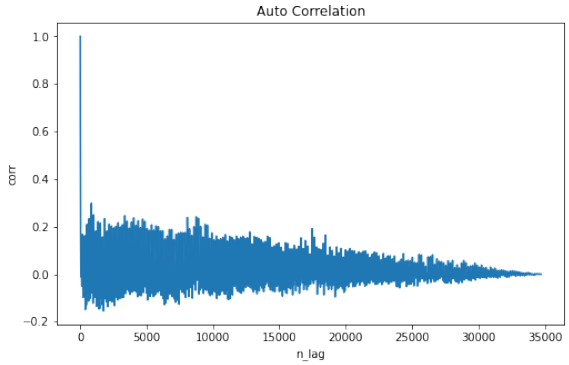
Dep. Variable:	power	R-squared:	0.848
Model:	OLS	Adj. R-squared:	0.847
Method:	Least Squares	F-statistic:	3.860e+04
Date:	Sun, 22 Jan 2023	Prob (F-statistic):	0.00
Time:	19:25:04	Log-Likelihood:	-2.5626e+05
No. Observations:	34728	AIC:	5.125e+05
Df Residuals:	34722	BIC:	5.126e+05
Df Model:	5		
Covariance Type:	nonrobust		

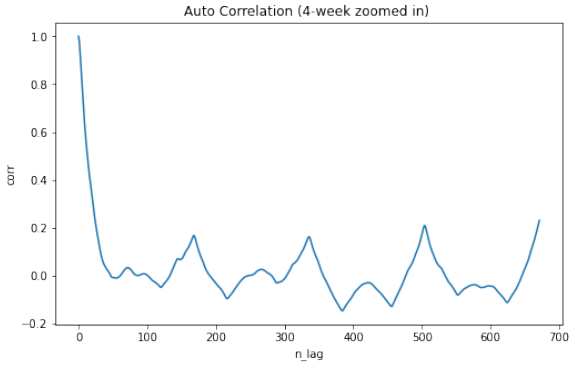
	coef	std err	t	P>|t|	[0.025	0.975]
const	200.8402	14.046	14.298	0.000	173.309	228.371
temp_hot	3.2508	0.217	14.983	0.000	2.826	3.676
temp_cold	-5.6865	0.379	-15.005	0.000	-6.429	-4.944
power_lag_1_day	0.5637	0.004	152.597	0.000	0.556	0.571
power_lag_1_week	0.2415	0.004	60.139	0.000	0.234	0.249
power_lag_2_week	0.1565	0.004	40.465	0.000	0.149	0.164

Omnibus:	2229.659	Durbin-Watson:	0.036
Prob(Omnibus):	0.000	Jarque-Bera (JB):	7850.238
Skew:	0.262	Prob(JB):	0.00
Kurtosis:	5.270	Cond. No.	6.72e+04

```
plt_acf(res)
```

```
/usr/local/lib/python3.8/dist-packages/statsmodels/tsa/stattools.py:667: FutureWarning: fft=T
  warnings.warn(
```

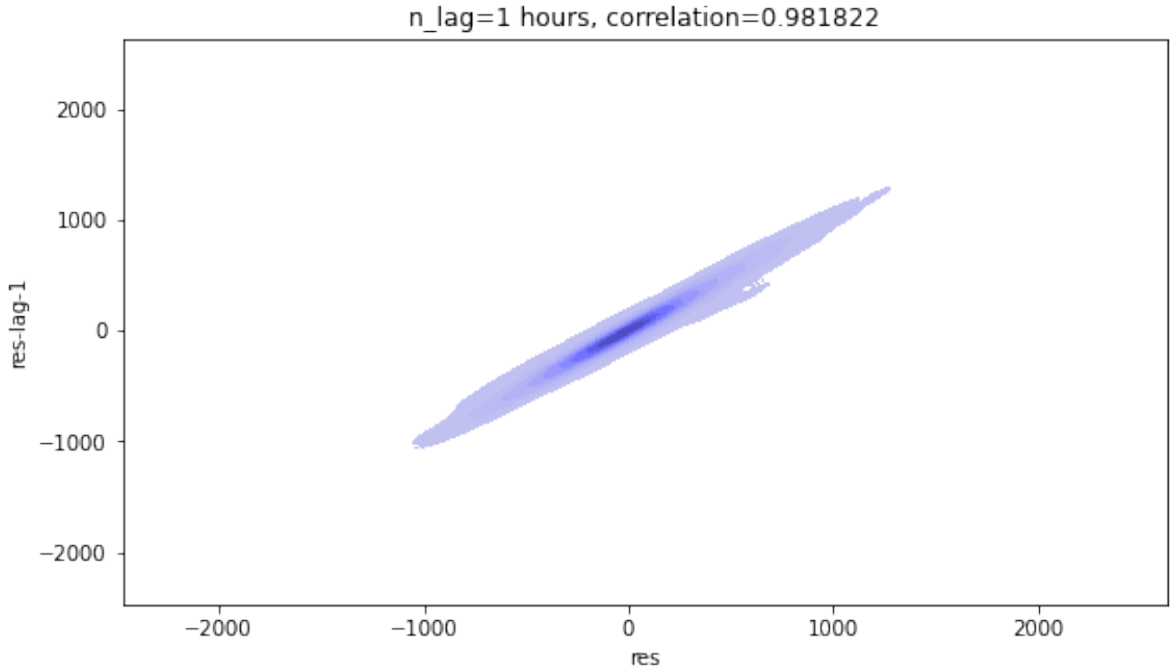




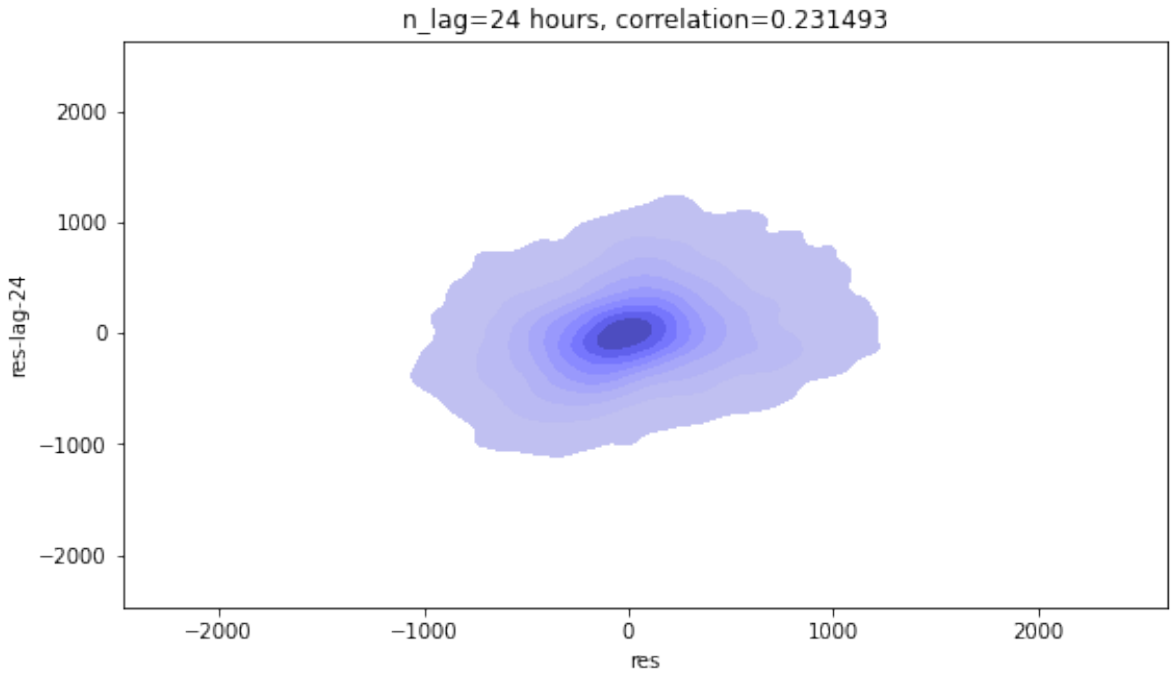
day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.00	0.98	0.94	0.90	0.85	0.80	0.75	0.71	0.67	0.63	0.59	0.56	0.53	0.50	0.4
1	0.23	0.21	0.20	0.18	0.16	0.14	0.13	0.11	0.10	0.08	0.07	0.06	0.05	0.05	0.0
2	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.00	-0.00	0.0
3	0.03	0.03	0.03	0.03	0.02	0.02	0.02	0.01	0.01	0.01	0.00	0.00	0.00	-0.00	-0
4	0.01	0.01	0.00	0.00	0.00	-0.00	-0.00	-0.01	-0.01	-0.01	-0.02	-0.02	-0.02	-0.02	-0
5	-0.05	-0.05	-0.04	-0.04	-0.04	-0.03	-0.03	-0.03	-0.02	-0.02	-0.02	-0.01	-0.01	-0.00	0.0
6	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.08	0.08	0.09	0.09	0.10	0.10	0.1
7	0.17	0.16	0.15	0.14	0.13	0.12	0.11	0.10	0.09	0.09	0.08	0.07	0.07	0.06	0.0
8	-0.00	-0.01	-0.01	-0.02	-0.02	-0.02	-0.03	-0.03	-0.03	-0.04	-0.04	-0.04	-0.05	-0.05	-0
9	-0.10	-0.10	-0.09	-0.09	-0.09	-0.08	-0.08	-0.08	-0.07	-0.07	-0.06	-0.06	-0.06	-0.05	-0
10	-0.01	-0.01	-0.01	-0.01	-0.00	-0.00	-0.00	-0.00	-0.00	-0.00	0.00	0.00	0.00	0.00	0.0
11	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.01	0.01	0.01	0.01	0.0
12	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0.02	-0.02	-0.02	-0.02	-0.01	-0.01	-0
13	0.05	0.05	0.05	0.05	0.06	0.06	0.06	0.07	0.07	0.08	0.08	0.09	0.09	0.10	0.1
14	0.16	0.16	0.15	0.14	0.13	0.12	0.11	0.10	0.09	0.08	0.08	0.07	0.06	0.06	0.0
15	-0.02	-0.03	-0.03	-0.04	-0.05	-0.05	-0.06	-0.06	-0.07	-0.08	-0.08	-0.09	-0.09	-0.10	-0
16	-0.15	-0.15	-0.14	-0.14	-0.13	-0.13	-0.12	-0.12	-0.11	-0.11	-0.11	-0.10	-0.10	-0.10	-0
17	-0.05	-0.05	-0.05	-0.04	-0.04	-0.04	-0.04	-0.04	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03	-0
18	-0.05	-0.05	-0.06	-0.06	-0.06	-0.07	-0.07	-0.07	-0.07	-0.08	-0.08	-0.09	-0.09	-0.09	-0
19	-0.13	-0.13	-0.12	-0.11	-0.11	-0.10	-0.09	-0.09	-0.08	-0.08	-0.07	-0.07	-0.06	-0.05	-0
20	0.02	0.03	0.03	0.04	0.04	0.05	0.05	0.06	0.07	0.07	0.08	0.09	0.10	0.10	0.1

day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
21	0.21	0.20	0.19	0.18	0.17	0.16	0.15	0.13	0.12	0.11	0.11	0.10	0.09	0.09	0.0
22	0.03	0.03	0.02	0.02	0.01	0.00	-0.00	-0.01	-0.01	-0.02	-0.02	-0.03	-0.03	-0.04	-0
23	-0.08	-0.08	-0.08	-0.08	-0.07	-0.07	-0.07	-0.06	-0.06	-0.06	-0.06	-0.05	-0.05	-0.05	-0
24	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04	-0.05	-0.05	-0.05	-0.05	-0.05	-0.05	-0.05	-0.05	-0
25	-0.04	-0.04	-0.05	-0.05	-0.05	-0.05	-0.05	-0.06	-0.06	-0.06	-0.07	-0.07	-0.07	-0.08	-0
26	-0.11	-0.11	-0.11	-0.10	-0.10	-0.09	-0.09	-0.08	-0.08	-0.08	-0.07	-0.07	-0.06	-0.06	-0
27	0.02	0.02	0.03	0.04	0.04	0.05	0.06	0.07	0.07	0.09	0.10	0.11	0.11	0.12	0.1
28	0.23	0.23	0.22	0.20	0.19	0.18	0.17	0.16	0.15	0.14	0.13	0.12	0.11	0.11	0.1
29	0.03	0.03	0.02	0.01	0.01	-0.00	-0.01	-0.01	-0.02	-0.03	-0.03	-0.04	-0.05	-0.05	-0

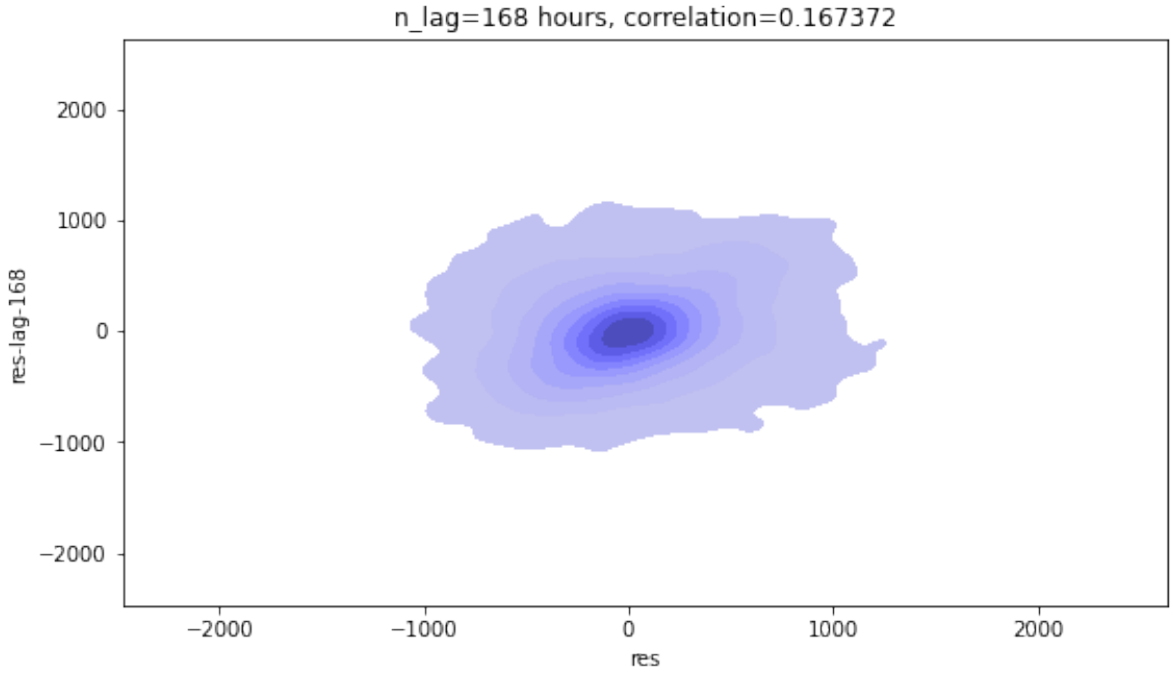
```
plt_residual_lag(res, 1)
```



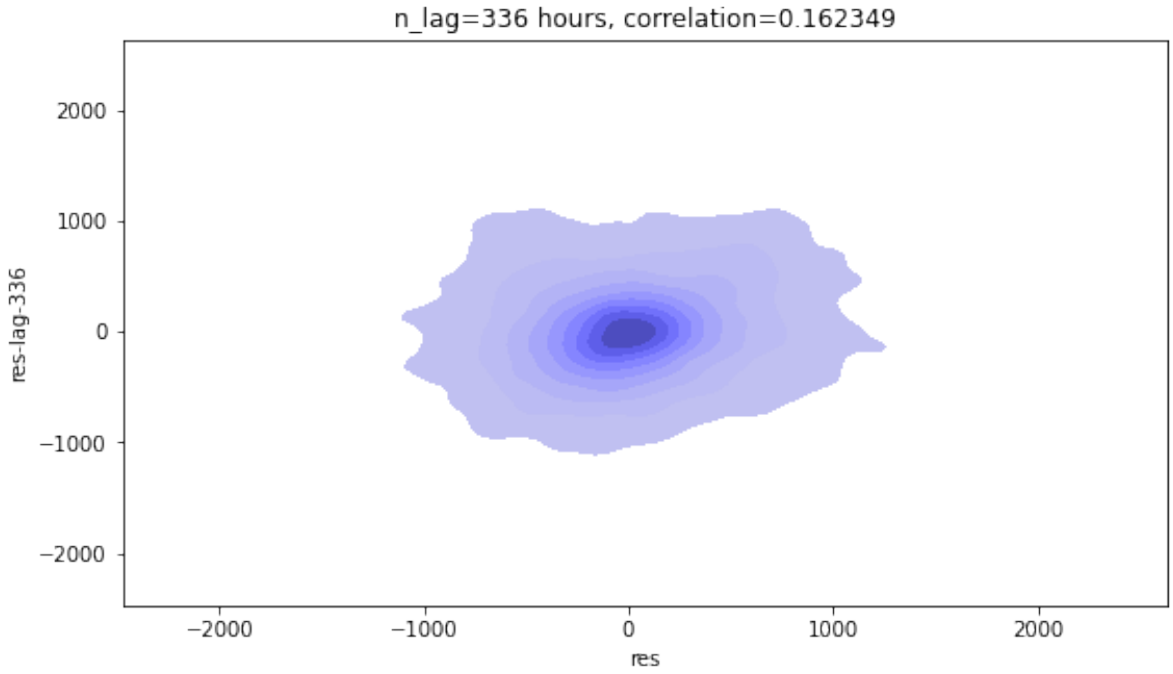

```
plt_residual_lag(res, 24)
```




```
plt_residual_lag(res, 24*7)
```



```
plt_residual_lag(res, 24*7*2)
```



7 Remark

We saw that with 2-week-lag feature, the R^2 only increased a little. The model summary seems still good so we could keep it. However, from the viewpoint of interpretation I may remove it.

One may also notice that the 1-day-lag correlation becomes bigger although 1-day-lag feature is already in the model. It is probably because of the multicollinearity between the lag features.

The following table shows the correlation between lag features.

```
df[['power_lag_1_day','power_lag_1_week', 'power_lag_2_week' ]].corr()
```

	power_lag_1_day	power_lag_1_week	power_lag_2_week
power_lag_1_day	1.000000	0.768394	0.745817
power_lag_1_week	0.768394	1.000000	0.819955
power_lag_2_week	0.745817	0.819955	1.000000

Part II

Logistic regression

8 Logistic regression

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.

8.1 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.

8.1.1 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.

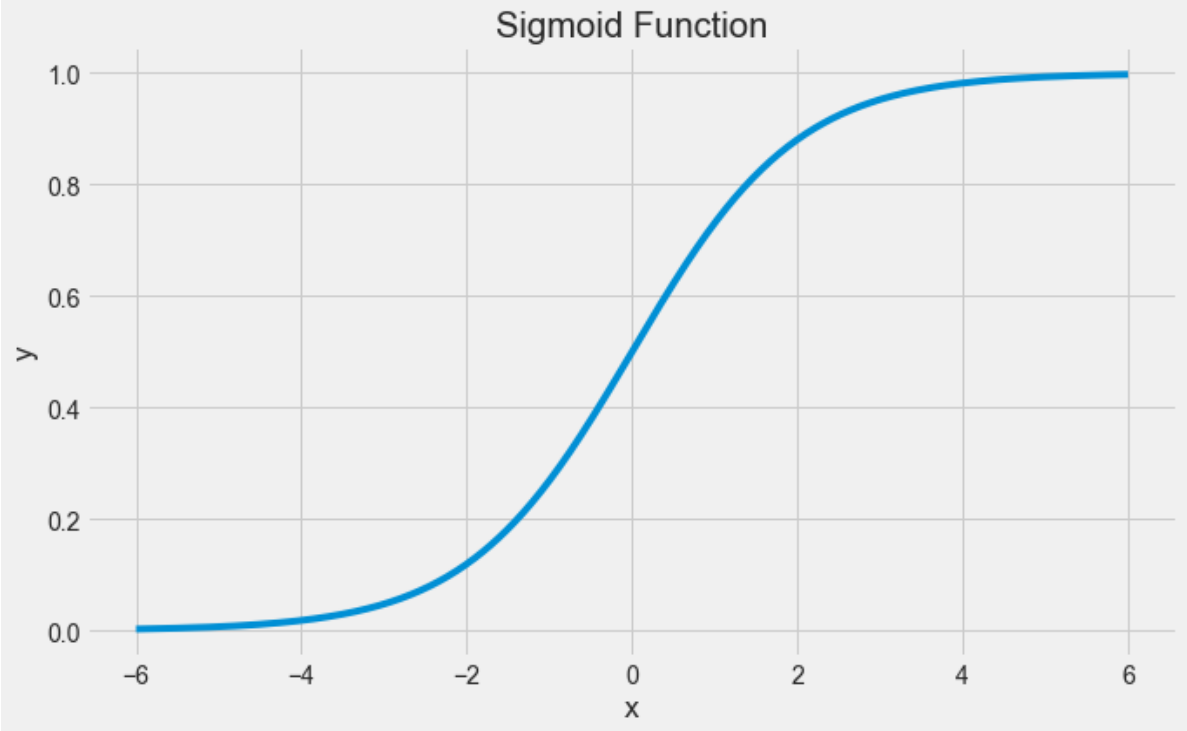
$$\frac{1}{1+e^{-x}}$$

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
```

```
%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.

8.1.2 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.

8.1.3 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).

8.2 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?"

Let us see the use of `statsmodels` for logistic regression. We'll see scikit-learn later in the course, when we learn methods that focus on prediction.

8.3 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.

```
train = pd.read_csv('./Datasets/Social_Network_Ads_train.csv') #Develop the model on train
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

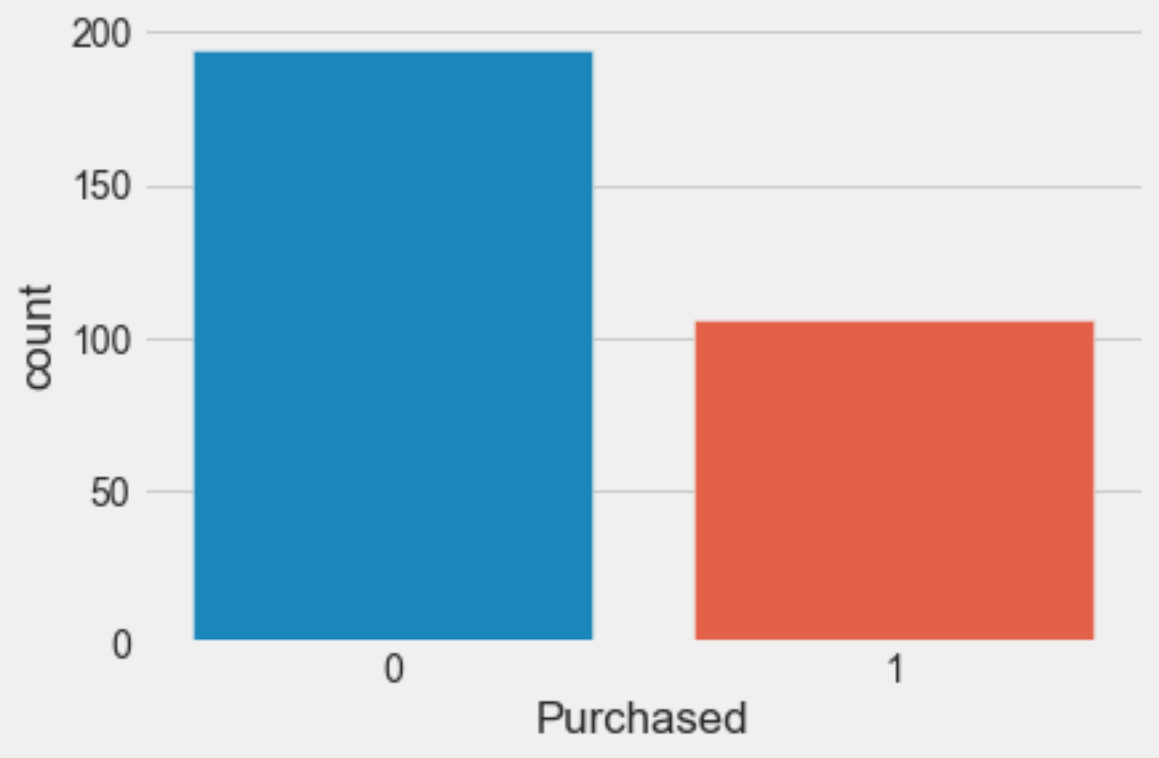
8.3.1 Examining the Distribution of the Target Column

Make sure our target is not severely imbalanced.

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

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

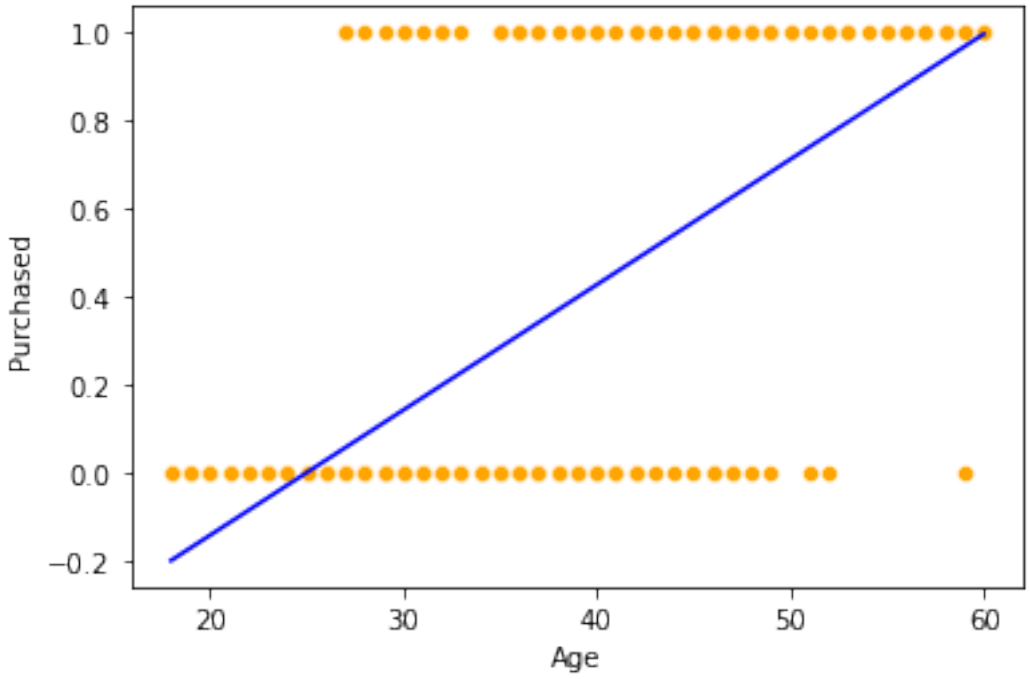
```
sns.countplot(x = 'Purchased',data = train);
```



Let us try to fit a linear regression model, instead of logistic regression. We fit a linear regression model to predict probability of purchase based on age.

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

```
<AxesSubplot:xlabel='Age', ylabel='Purchased'>
```



Note the issues with the linear regression model:

1. The regression line goes below 0 and over 1. However, probability of purchase must be in [0,1].
2. The linear regression model does not seem to fit the data well.

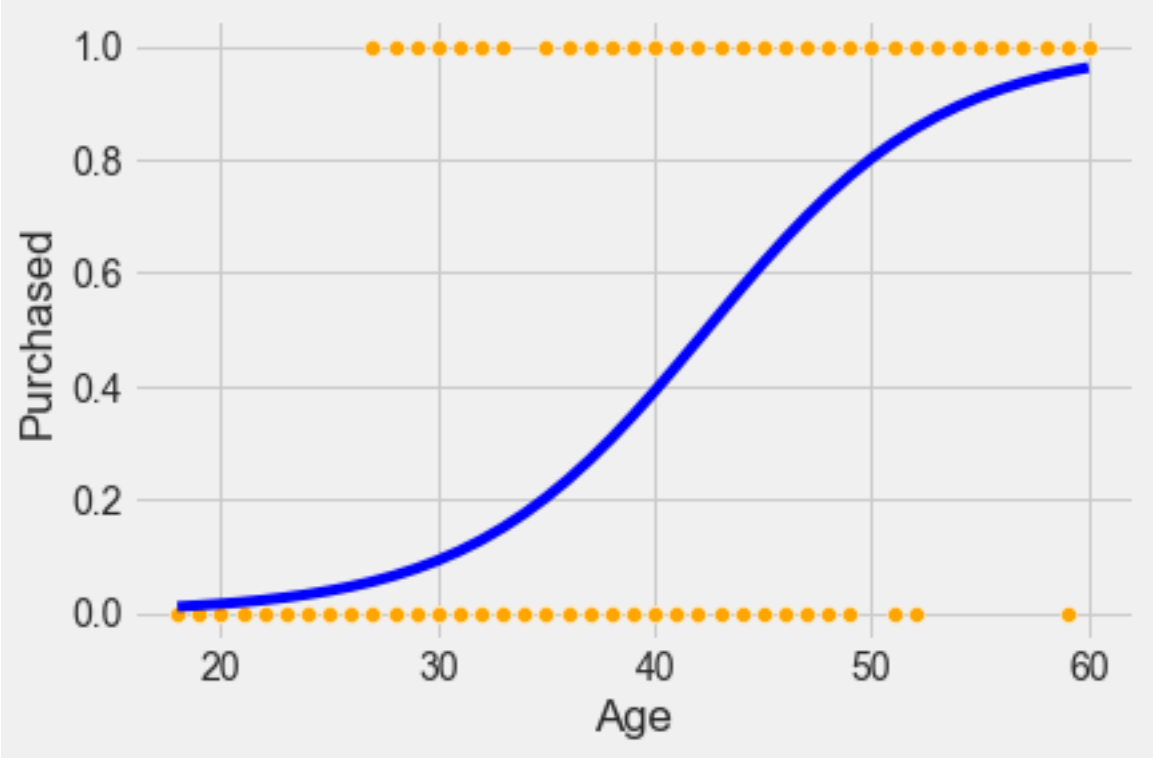
8.3.2 Fitting the logistic regression model

Now, let us fit a logistic regression model to predict probability of purchase based on `Age`.

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

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

```
<AxesSubplot:xlabel='Age', ylabel='Purchased'>
```



As logistic regression uses the sigmoid function, the probability stays in [0,1]. Also, it seems to better fit the points as compared to linear regression.

```
logit_model.summary()
```

Table 8.2: Logit Regression Results

Dep. Variable:	Purchased	No. Observations:	300
Model:	Logit	Df Residuals:	298
Method:	MLE	Df Model:	1
Date:	Tue, 19 Apr 2022	Pseudo R-squ.:	0.3378
Time:	16:46:02	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

Interpret the coefficient of age

For a unit increase in age, the log odds of purchase increase by 0.18, or the odds of purchase get multiplied by $\exp(0.18) = 1.2$

Is the increase in probability of purchase constant with a unit increase in age?

No, it depends on age.

Is gender associated with probability of purchase?

```
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
```

Table 8.4: Logit Regression Results

Dep. Variable:	Purchased	No. Observations:	300
Model:	Logit	Df Residuals:	298
Method:	MLE	Df Model:	1
Date:	Tue, 19 Apr 2022	Pseudo R-squ.:	0.001049
Time:	16:46:04	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

No, assuming a significance level of $\alpha = 5\%$, `Gender` is not associated with probability of default, as the p-value for `Male` is greater than 0.05.

8.4 Confusion matrix and classification accuracy

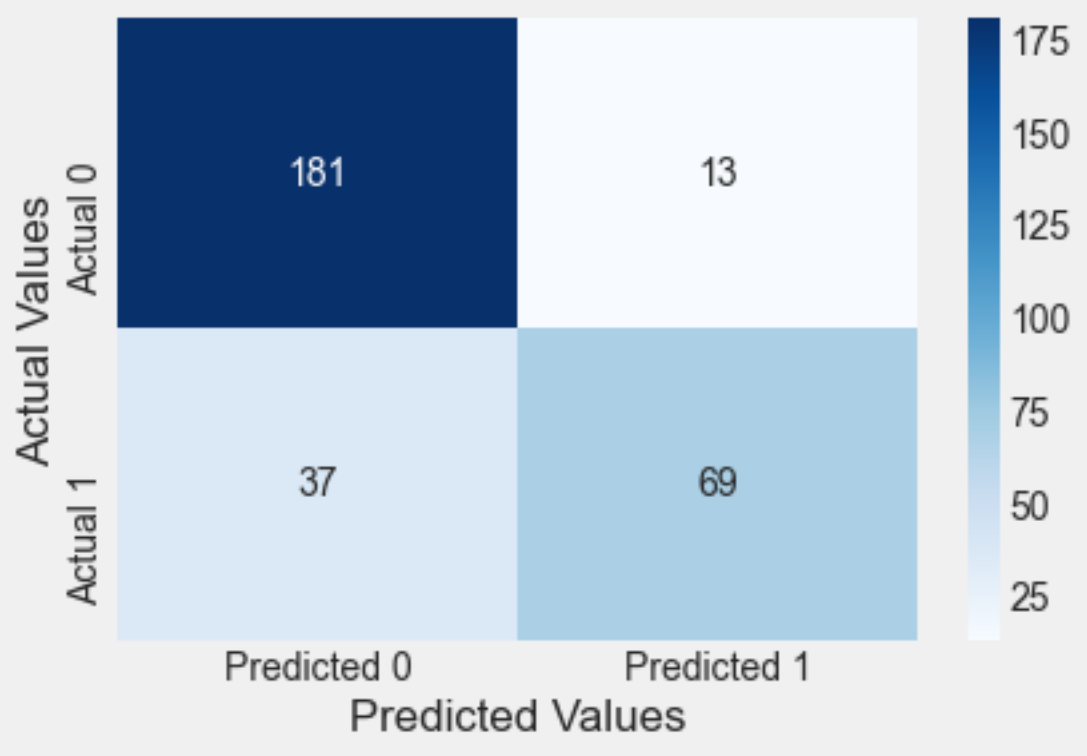
A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class.

```
#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))
```

Find the confusion matrix and classification accuracy of the model with `Age` as the predictor on training data.

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

```
Classification accuracy = 83.3%
```



Confusion matrix:

- Each row: actual class
- Each column: predicted class

First row: Non-purchasers, the negative class:

- 181 were correctly classified as Non-purchasers. **True negatives**.
- Remaining 13 were wrongly classified as Non-purchasers. **False positive**

Second row: Purchasers, the positive class:

- 37 were incorrectly classified as Non-purchasers. **False negatives**
- 69 were correctly classified Purchasers. **True positives**

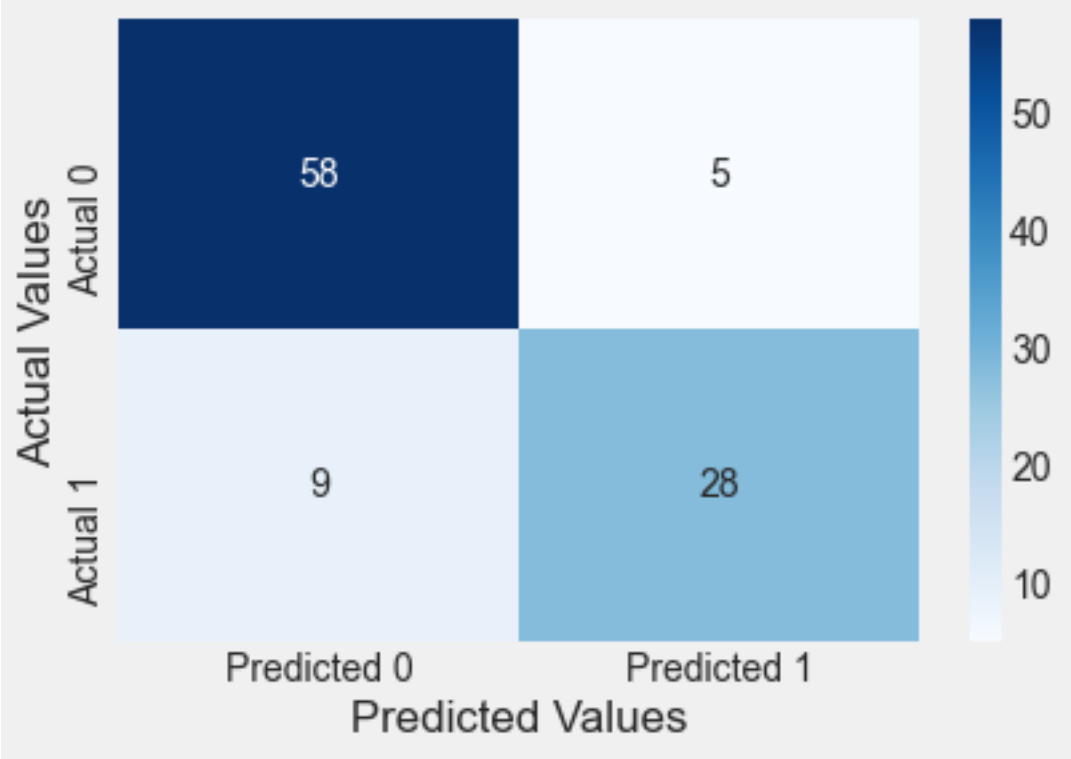
```
#Function to compute confusion matrix and prediction accuracy on test data
def confusion_matrix_test(data,actual_values,model,cutoff=0.5):
#Predict the values using the Logit model
    pred_values = model.predict(data)
# Specify the bins
    bins=np.array([0,cutoff,1])
```

```
#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'})
    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))
```

Find the confusion matrix and classification accuracy of the model with `Age` as the predictor on test data.

```
confusion_matrix_test(test,test.Purchased,logit_model)
```

```
Classification accuracy = 86.0%
```



The model classifies a bit more accurately on test data as compared to the training data, which

is a bit unusual. However, it shows that the model did not overfit on training data.

Include `EstimatedSalary` as a predictor in the above model

```
logit_model2 = sm.logit(formula = 'Purchased~Age+EstimatedSalary', data = train).fit()
logit_model2.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.358910
         Iterations 7
```

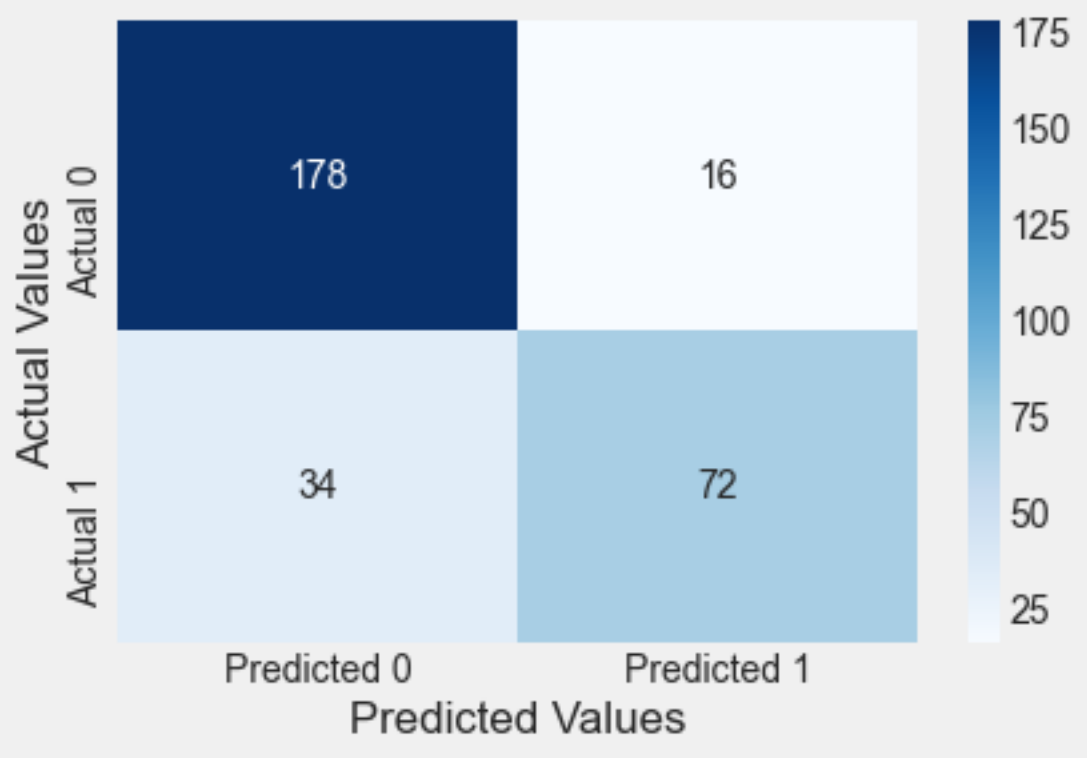
Table 8.6: Logit Regression Results

Dep. Variable:	Purchased	No. Observations:	300
Model:	Logit	Df Residuals:	297
Method:	MLE	Df Model:	2
Date:	Tue, 14 Feb 2023	Pseudo R-squ.:	0.4474
Time:	12:03:29	Log-Likelihood:	-107.67
converged:	True	LL-Null:	-194.85
Covariance Type:	nonrobust	LLR p-value:	1.385e-38

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-11.9432	1.424	-8.386	0.000	-14.735	-9.152
Age	0.2242	0.028	7.890	0.000	0.168	0.280
EstimatedSalary	3.48e-05	6.15e-06	5.660	0.000	2.27e-05	4.68e-05

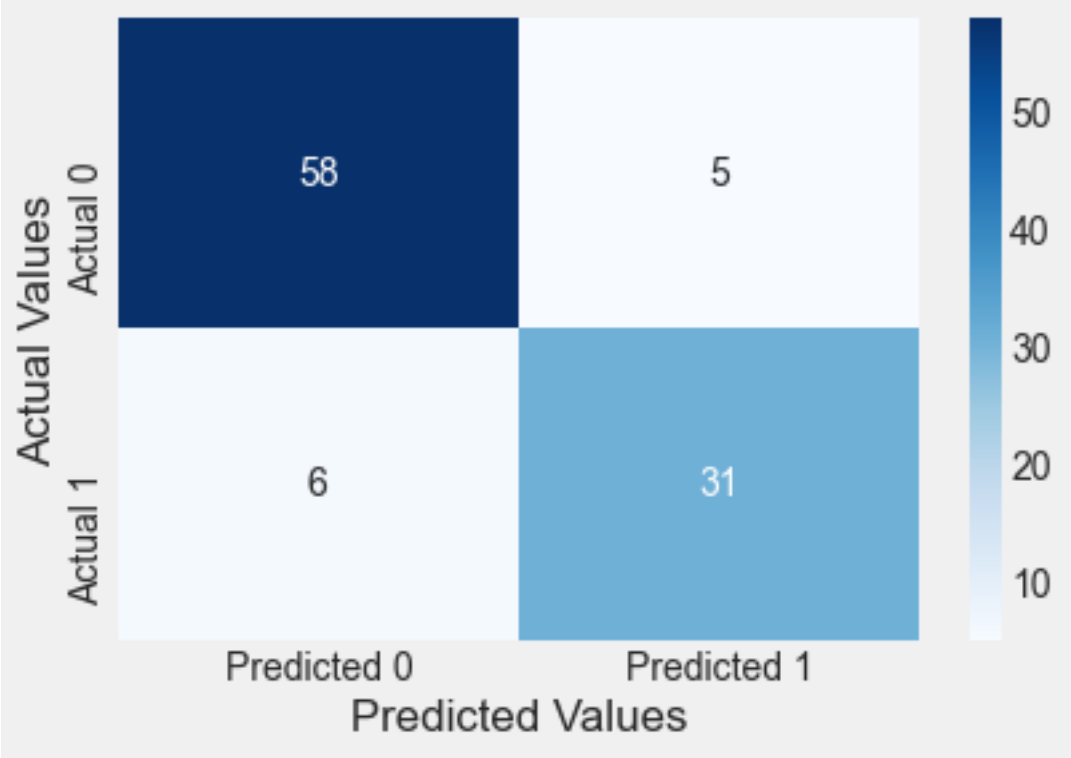
```
confusion_matrix_train(logit_model2)
```

```
Classification accuracy = 83.3%
```




```
confusion_matrix_test(test,test.Purchased,logit_model2)
```

```
Classification accuracy = 89.0%
```



The log likelihood of the model has increased, while also increasing the prediction accuracy on test data, which shows that the additional predictor is helping explain the response better, without overfitting the data.

Include `Gender` as a predictor in the above model

```
logit_model = sm.logit(formula = 'Purchased~Age+EstimatedSalary+Gender', data = train).fit
logit_model.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.357327
         Iterations 7
```

Table 8.8: Logit Regression Results

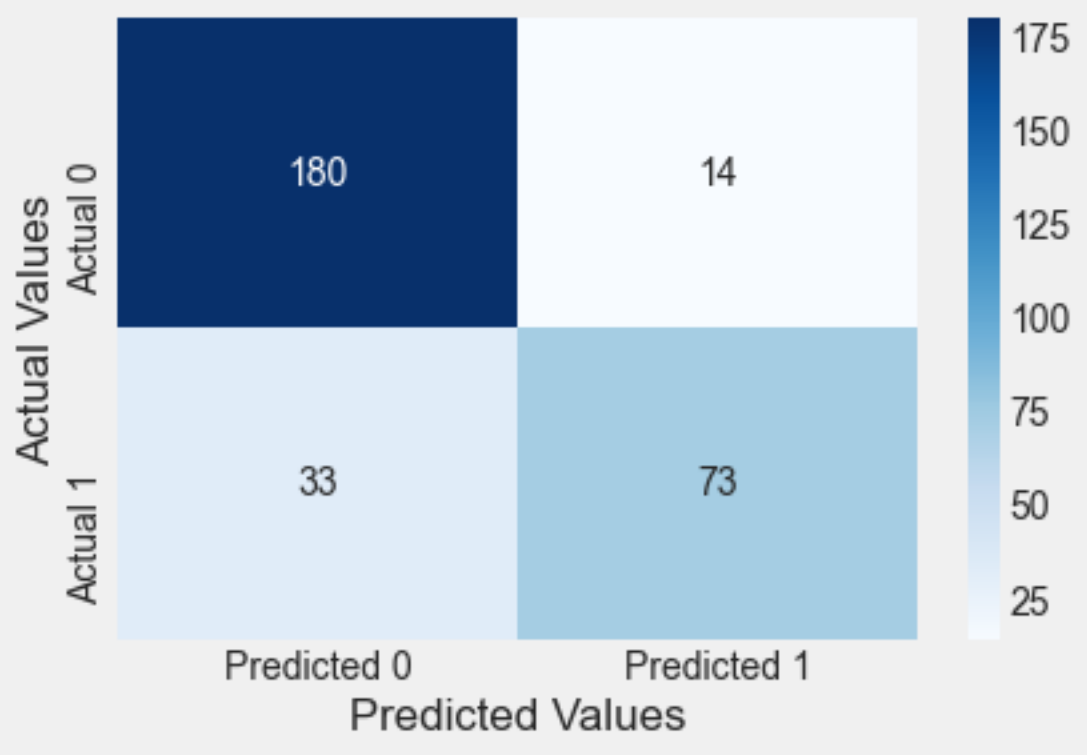
Dep. Variable:	Purchased	No. Observations:	300
Model:	Logit	Df Residuals:	296
Method:	MLE	Df Model:	3
Date:	Tue, 14 Feb 2023	Pseudo R-squ.:	0.4498

Time:	12:17:28	Log-Likelihood:	-107.20
converged:	True	LL-Null:	-194.85
Covariance Type:	nonrobust	LLR p-value:	9.150e-38

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-12.2531	1.478	-8.293	0.000	-15.149	-9.357
Gender[T.Male]	0.3356	0.346	0.970	0.332	-0.342	1.013
Age	0.2275	0.029	7.888	0.000	0.171	0.284
EstimatedSalary	3.494e-05	6.17e-06	5.666	0.000	2.29e-05	4.7e-05

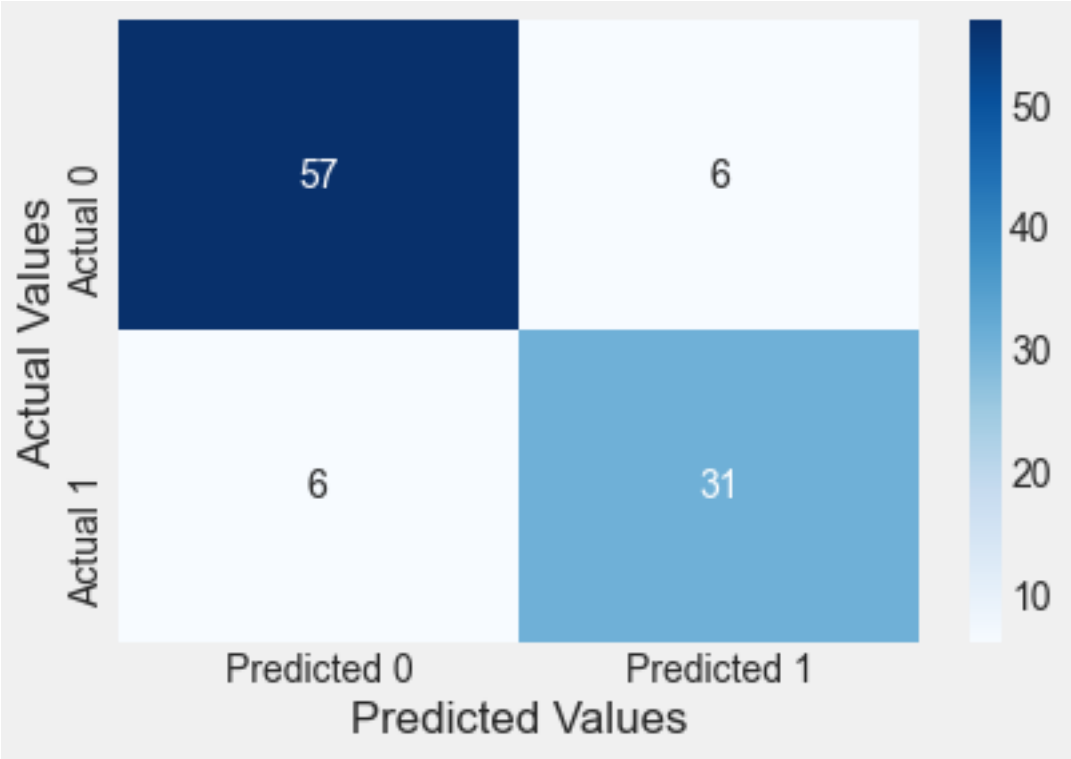
```
confusion_matrix_train(logit_model)
```

```
Classification accuracy = 84.3%
```




```
confusion_matrix_test(test,test.Purchased,logit_model)
```

```
Classification accuracy = 88.0%
```



`Gender` is a statistically insignificant predictor, and including it slightly lowers the classification accuracy on test data. Note that the classification accuracy on training data will continue to increase on adding more predictors, irrespective of their relevance *(similar to the idea of RSS on training data in linear regression)*.

Is there a residual in logistic regression?

No, since the response is assumed to have a Bernoulli distribution, instead of a normal distribution.

Is the odds ratio for a unit increase in a predictor X_j, a constant (assuming that the rest of the predictors are held constant)?

Yes, the odds ratio in this case will e^{β_j}

8.5 Variable transformations in logistic regression

Read the dataset *diabetes.csv* that contains if a person has diabetes (`Outcome = 1`) based on health parameters such as BMI, blood pressure, age etc. Develop a model to predict the

probability of a person having diabetes based on their age.

```
data = pd.read_csv('./Datasets/diabetes.csv')
```

```
data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

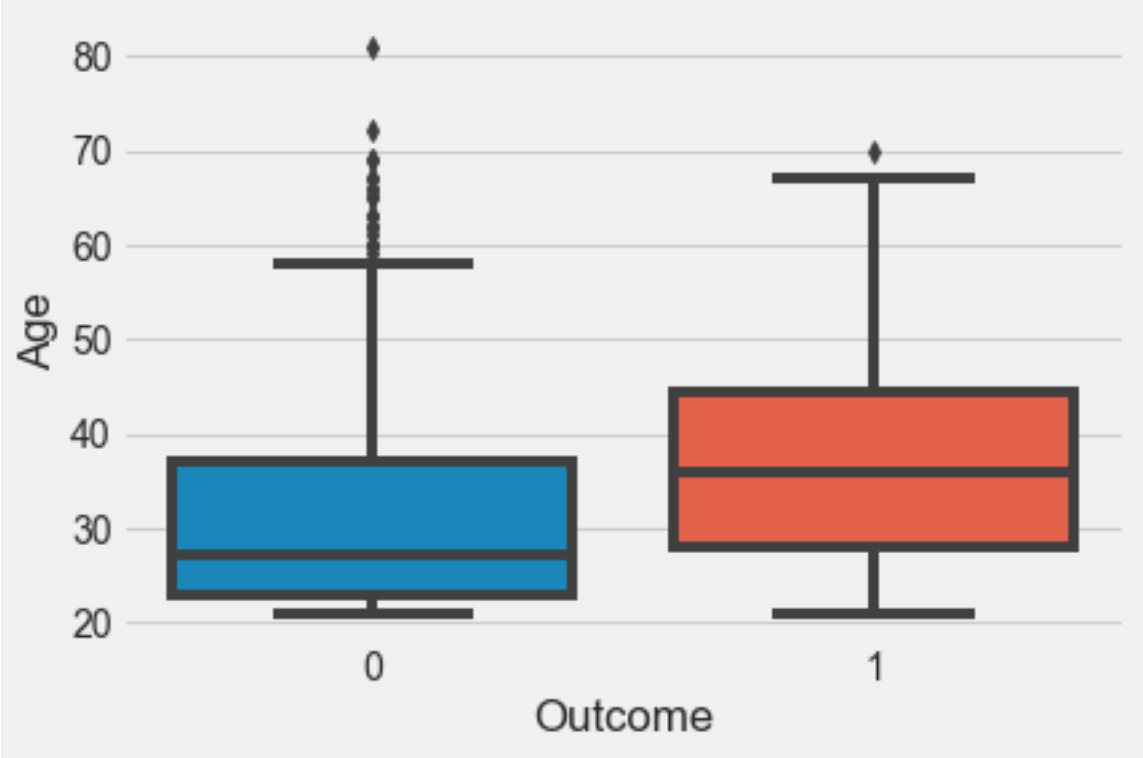
Randomly select 80% of the observations to create a training dataset. Create a test dataset with the remaining 20% observations.

```
#Creating training and test datasets
np.random.seed(2)
train = data.sample(round(data.shape[0]*0.8))
test = data.drop(train.index)
```

Does `Age` seem to distinguish `Outcome` levels?

```
sns.boxplot(x = 'Outcome', y = 'Age', data = train)
```

```
<AxesSubplot:xlabel='Outcome', ylabel='Age'>
```

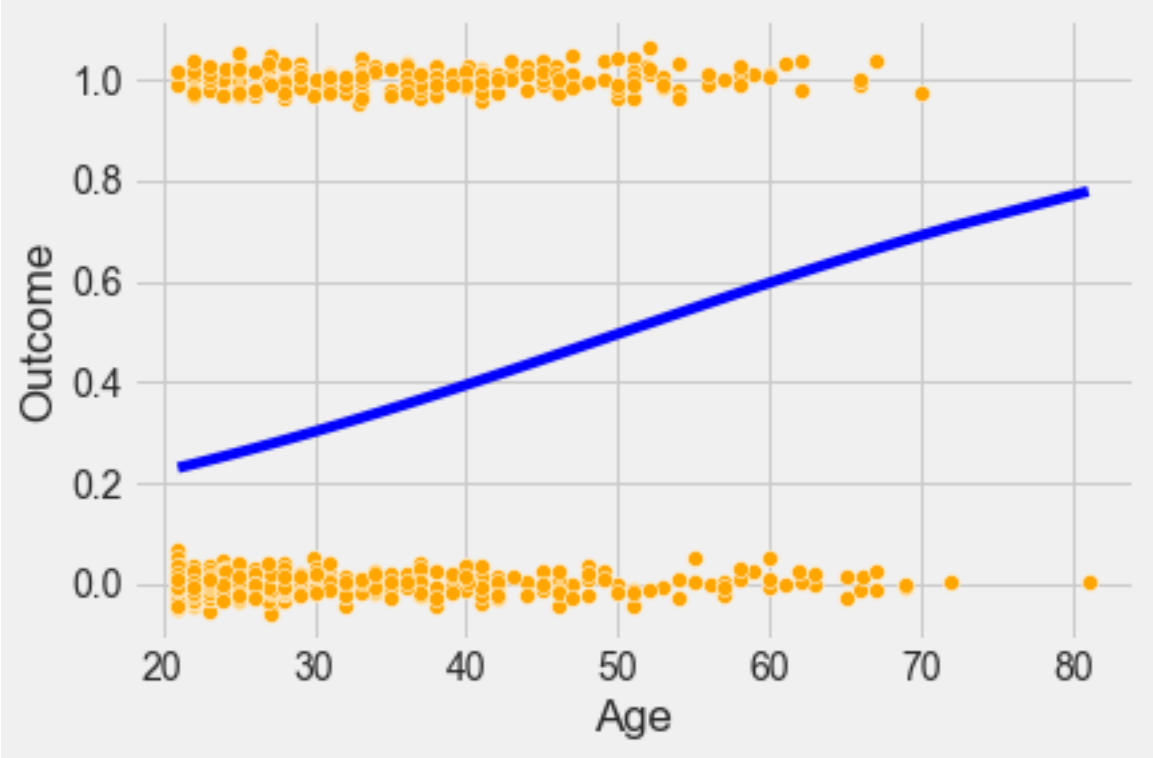


Yes it does!

Develop and visualize a logistic regression model to predict `Outcome` using `Age`.

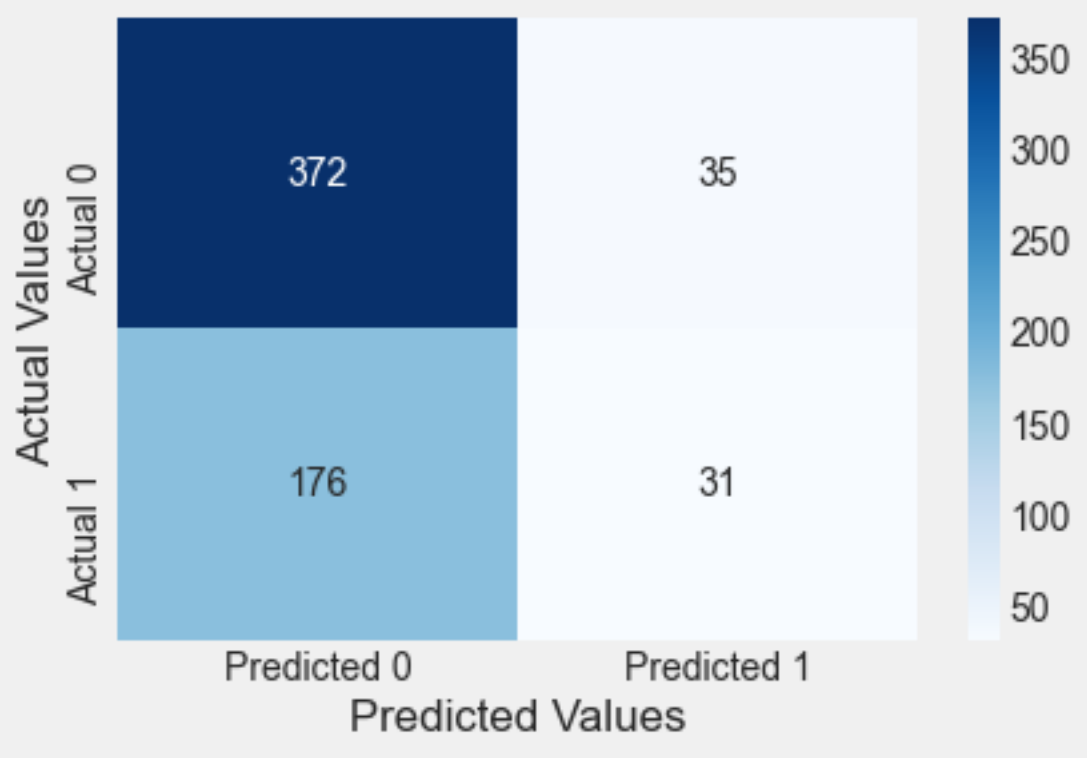
```
#Jittering points to better see the density of points in any given region of the plot
def jitter(values,j):
    return values + np.random.normal(j,0.02,values.shape)
sns.scatterplot(x = jitter(train.Age,0), y = jitter(train.Outcome,0), data = train, color
logit_model = sm.logit(formula = 'Outcome~Age', data = train).fit()
sns.lineplot(x = 'Age', y= logit_model.predict(train), data = train, color = 'blue')
print(logit_model.llf) #Printing the log likelihood to compare it with the next model we b
```

```
Optimization terminated successfully.
         Current function value: 0.612356
         Iterations 5
-375.9863802089716
```




```
confusion_matrix_train(logit_model)
```

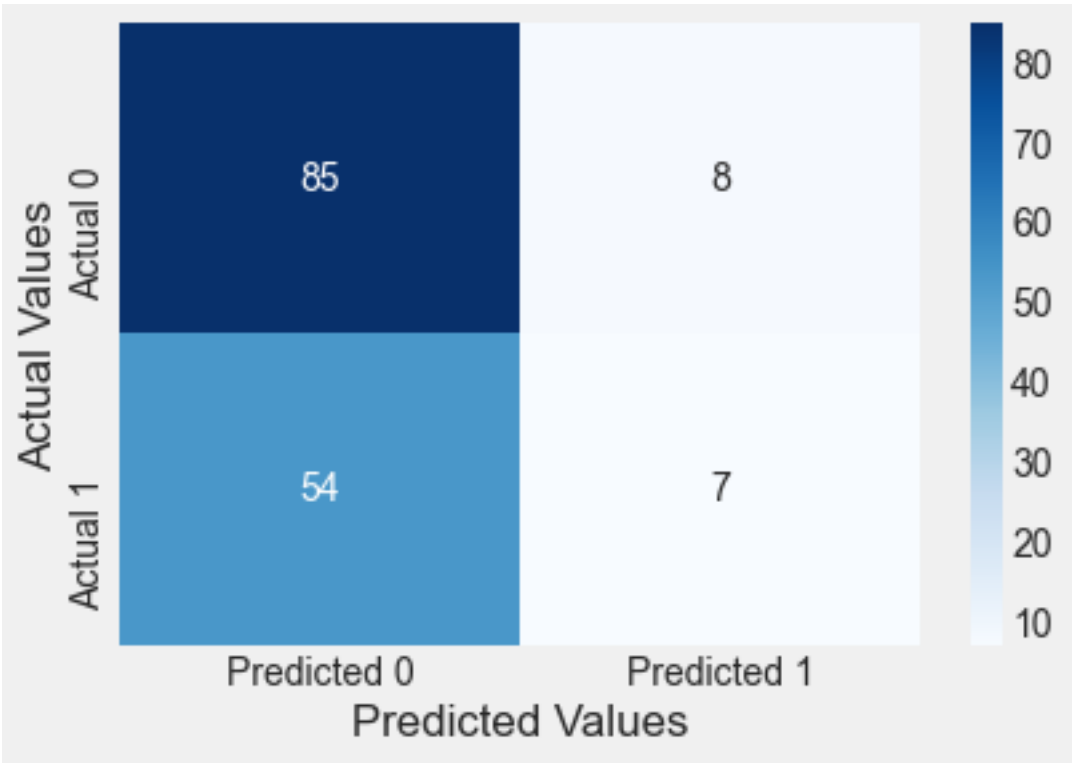
```
Classification accuracy = 65.6%
```



Classification accuracy on train data = 66%

```
confusion_matrix_test(test,test.Outcome,logit_model)
```

```
Classification accuracy = 59.7%
```



Classification accuracy on test data = 60%

Can a tranformation of `Age` provide a more accurate model?

Let us visualize how the probability of people having diabetes varies with `Age`. We will bin `Age` to get the percentage of people having diabetes within different `Age` bins.

```
#Binning Age
binned_age = pd.qcut(train['Age'],11,retbins=True)
train['age_binned'] = binned_age[0]
```

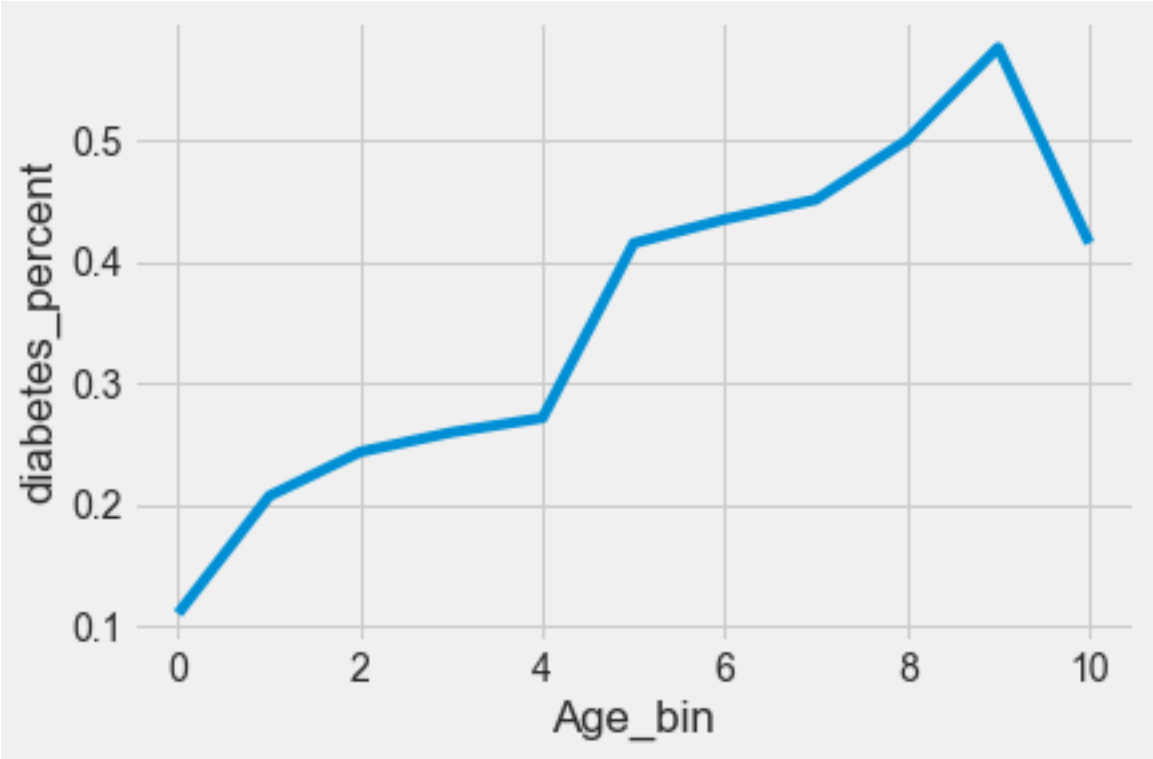
```
#Finding percentage of people having diabetes in each Age bin
age_data = train.groupby('age_binned')['Outcome'].agg([('diabetes_percent','mean'),('nobs'
age_data
```

	age_binned	diabetes_percent	nobs
0	(20.999, 22.0]	0.110092	109
1	(22.0, 23.0]	0.206897	29
2	(23.0, 25.0]	0.243243	74
3	(25.0, 26.0]	0.259259	27

	age_binned	diabetes_percent	nobs
4	(26.0, 28.0]	0.271186	59
5	(28.0, 31.0]	0.415094	53
6	(31.0, 35.0]	0.434783	46
7	(35.0, 39.0]	0.450980	51
8	(39.0, 43.545]	0.500000	54
9	(43.545, 52.0]	0.576271	59
10	(52.0, 81.0]	0.415094	53

```
#Visualizing percentage of people having diabetes with increasing Age (or Age bins)
sns.lineplot(x = age_data.index, y= age_data['diabetes_percent'])
plt.xlabel('Age_bin')
```

```
Text(0.5, 0, 'Age_bin')
```



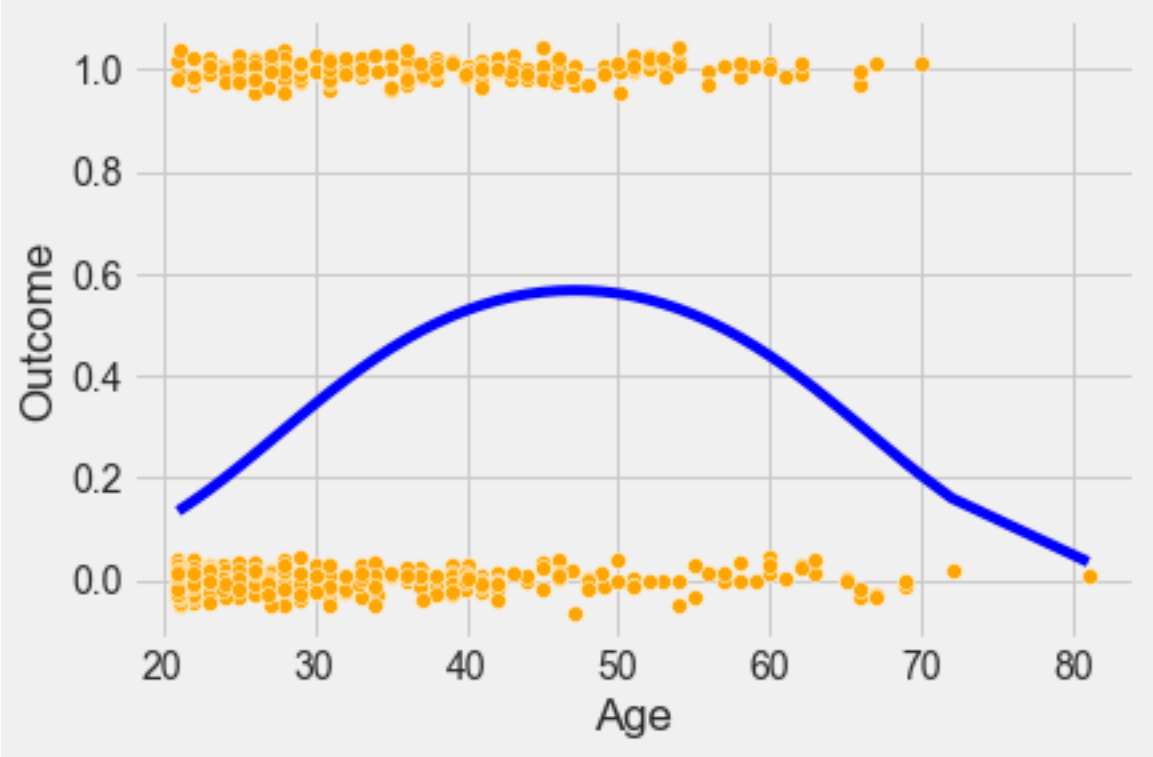
We observe that the probability of people having diabetes does **not** keep increasing monotonically with age. People with ages 52 and more have a lower probability of having diabetes than people in the immediately younger `Age` bin.

A quadratic transformation of `Age` may better fit the above trend

```
#Model with the quadratic transformation of Age
def jitter(values,j):
    return values + np.random.normal(j,0.02,values.shape)
sns.scatterplot(x = jitter(train.Age,0), y = jitter(train.Outcome,0), data = train, color
logit_model = sm.logit(formula = 'Outcome~Age+I(Age**2)', data = train).fit()
sns.lineplot(x = 'Age', y= logit_model.predict(train), data = train, color = 'blue')
logit_model.llf
```

```
Optimization terminated successfully.
         Current function value: 0.586025
         Iterations 6
```

```
-359.81925590230185
```




```
logit_model.summary()
```

Table 8.12: Logit Regression Results

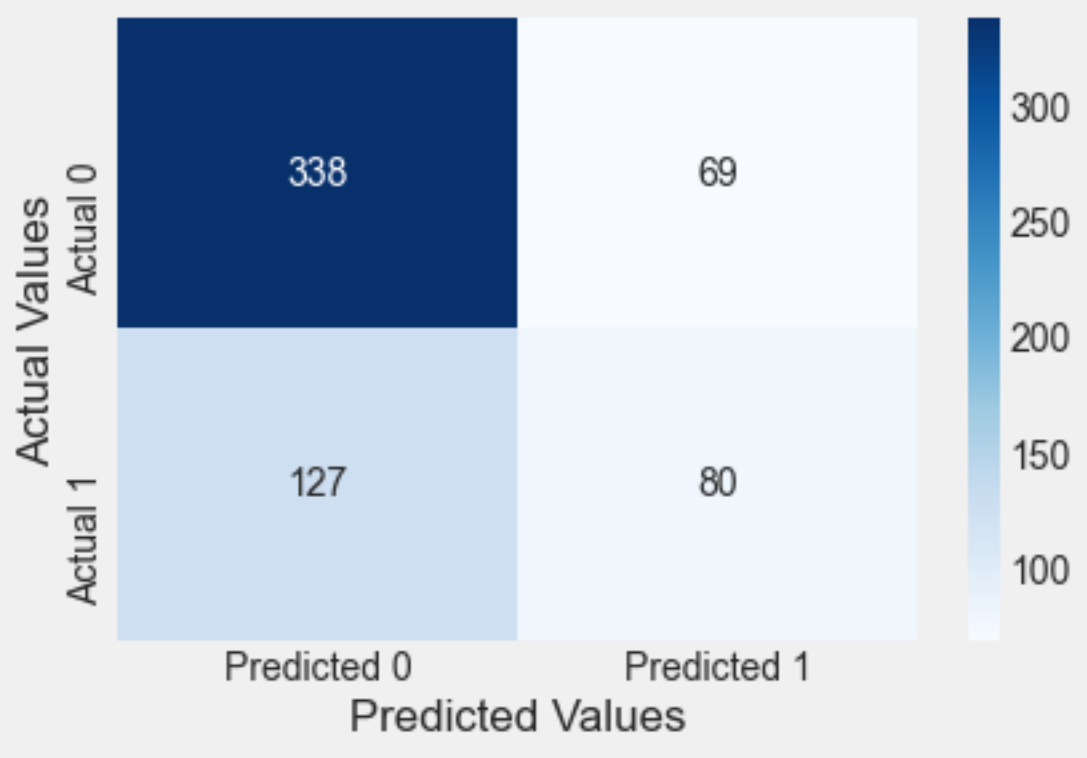
Dep. Variable:	Outcome	No. Observations:	614
Model:	Logit	Df Residuals:	611
Method:	MLE	Df Model:	2
Date:	Tue, 14 Feb 2023	Pseudo R-squ.:	0.08307
Time:	12:25:54	Log-Likelihood:	-359.82
converged:	True	LL-Null:	-392.42
Covariance Type:	nonrobust	LLR p-value:	6.965e-15

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-6.6485	0.908	-7.320	0.000	-8.429	-4.868
Age	0.2936	0.048	6.101	0.000	0.199	0.388
I(Age ** 2)	-0.0031	0.001	-5.280	0.000	-0.004	-0.002

The log likelihood of the model is higher and both the predictors are statistically significant indicating a better model fit. However, the model may also be overfitting. Let us check the model accuracy on test data.

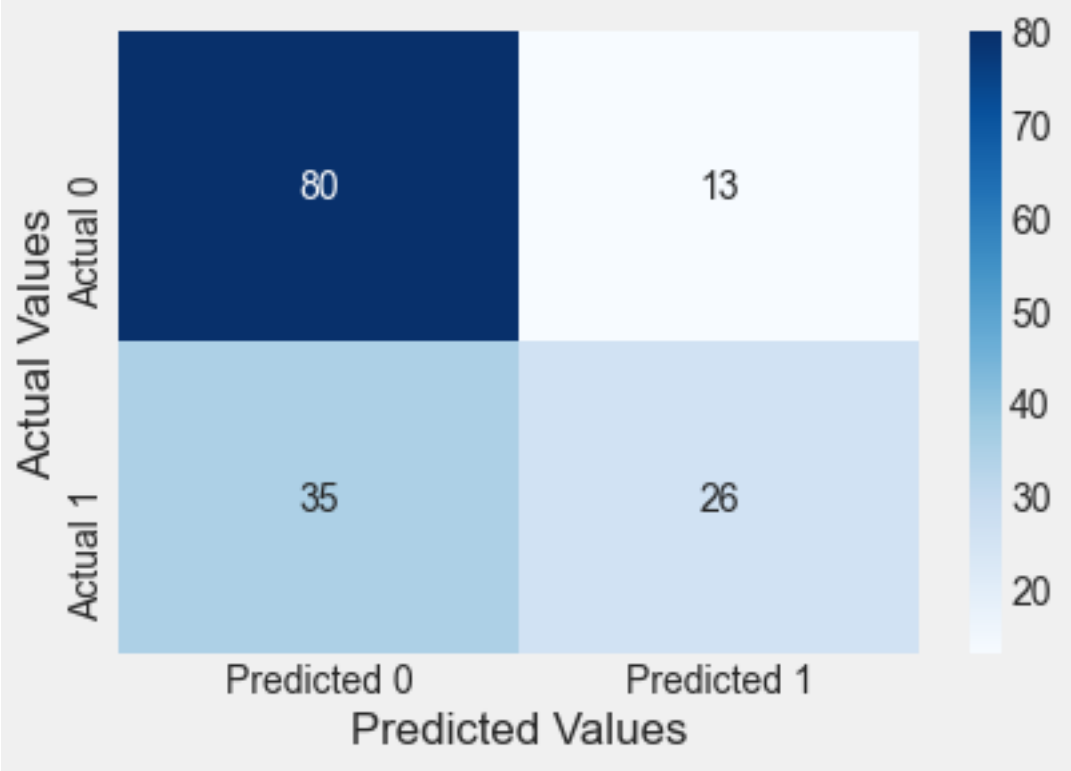
```
confusion_matrix_train(logit_model)
```

```
Classification accuracy = 68.1%
```




```
confusion_matrix_test(test,test.Outcome,logit_model)
```

```
Classification accuracy = 68.8%
```

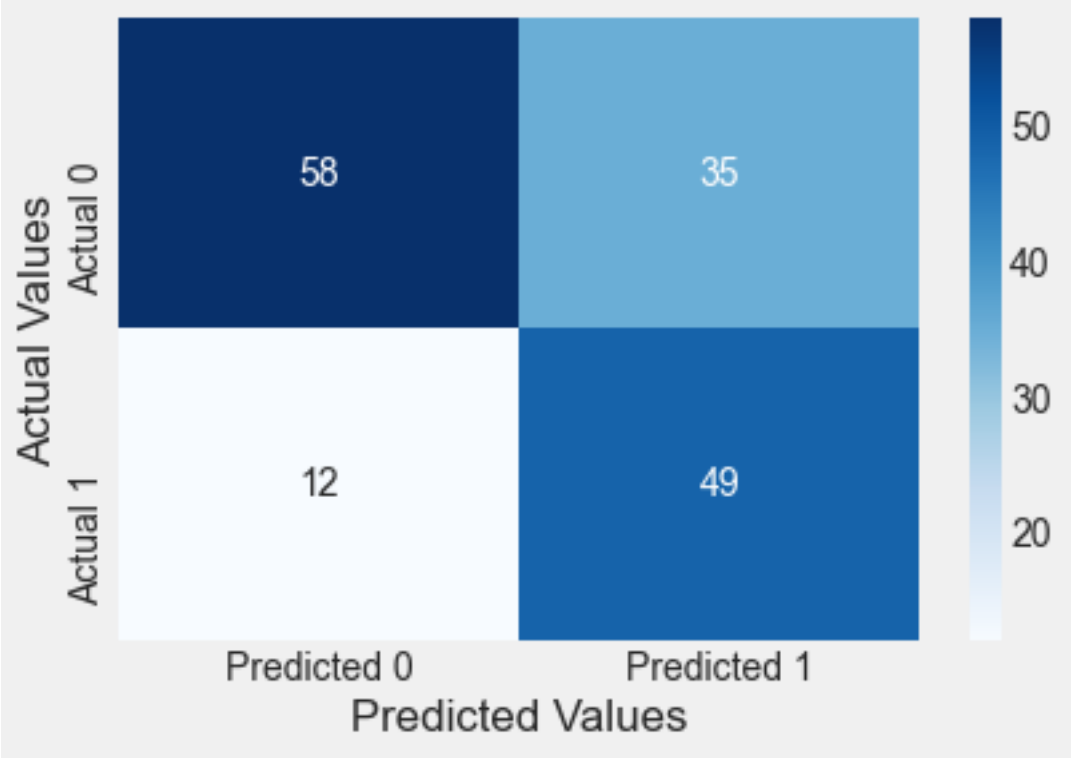


The classification accuracy on test data has increased to 69%. However, the number of *false positives* have increased. But in case of diabetes, *false negatives* are more concerning than *false positives*. This is because if a person has diabetes, and is told that they do not have diabetes, their condition may deteriorate. If a person does not have diabetes, and is told that they have diabetes, they may take unnecessary precautions or tests, but it will not be as harmful to the person as in the previous case. So, in this problem, we will be more focused on reducing the number of *false negatives*, instead of reducing the *false positives* or increasing the overall classification accuracy.

We can decrease the cutoff for classifying a person as having diabetes to reduce the number of false negatives.

```
#Reducing the cutoff for classifying a person as diabetic to 0.3 (instead of 0.5)
confusion_matrix_test(test,test.Outcome,logit_model,0.3)
```

```
Classification accuracy = 69.5%
```



Note that the changed cut-off reduced the number of *false negatives*, but at the cost of increasing the *false positives*. However, the stakeholders may prefer the reduced cut-off to be safer.

Is there another way to transform `Age`?

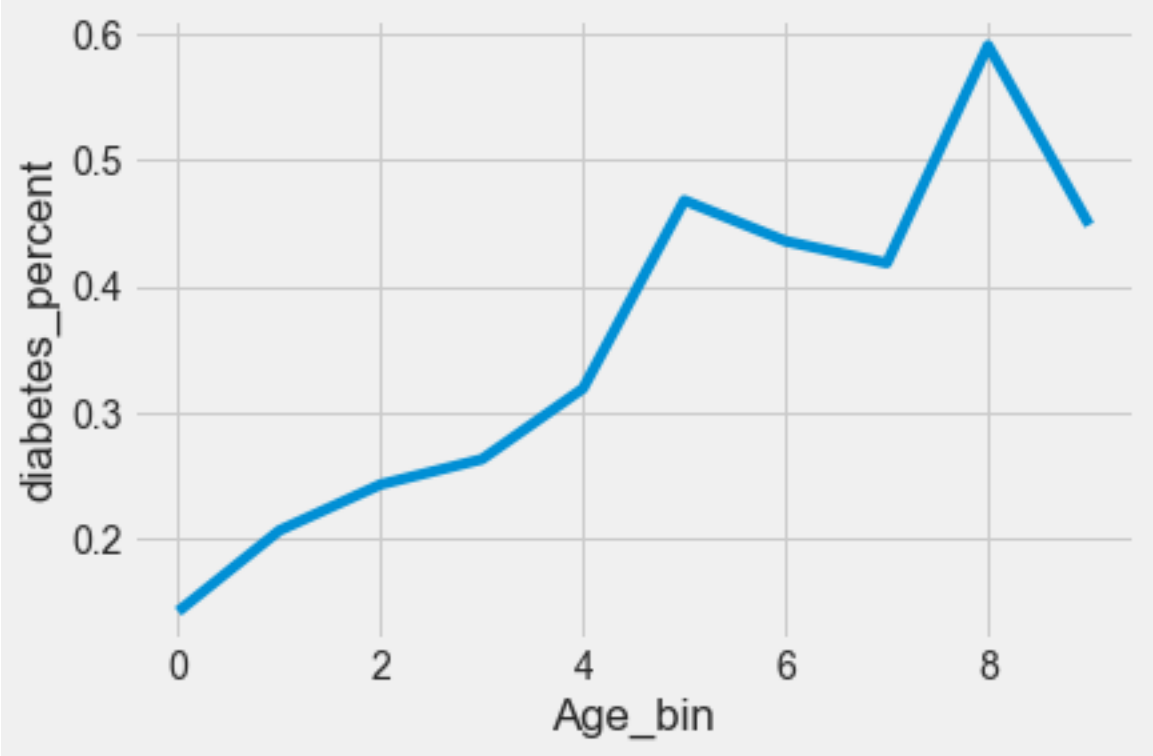
Yes, binning age into bins that have similar proportion of people with diabetes may provide a better model fit.

```
#Creating a function to bin age so that it can be applied to both the test and train datas
def var_transform(data):
    binned_age = pd.qcut(train['Age'],10,retbins=True)
    bins = binned_age[1]
    data['age_binned'] = pd.cut(data['Age'],bins = bins)
    dum = pd.get_dummies(data.age_binned,drop_first = True)
    dum.columns = ['age'+str(x) for x in range(1,len(bins)-1)]
    data = pd.concat([data,dum], axis = 1)
    return data
```

```
#Binning age using the function var_transform()
train = var_transform(train)
test = var_transform(test)
```

```
#Re-creating the plot of diabetes_percent vs age created earlier, just to check if the fun
age_data = train.groupby('age_binned')['Outcome'].agg([('diabetes_percent','mean'),('nobs'
sns.lineplot(x = age_data.index, y= age_data['diabetes_percent'])
plt.xlabel('Age_bin')
```

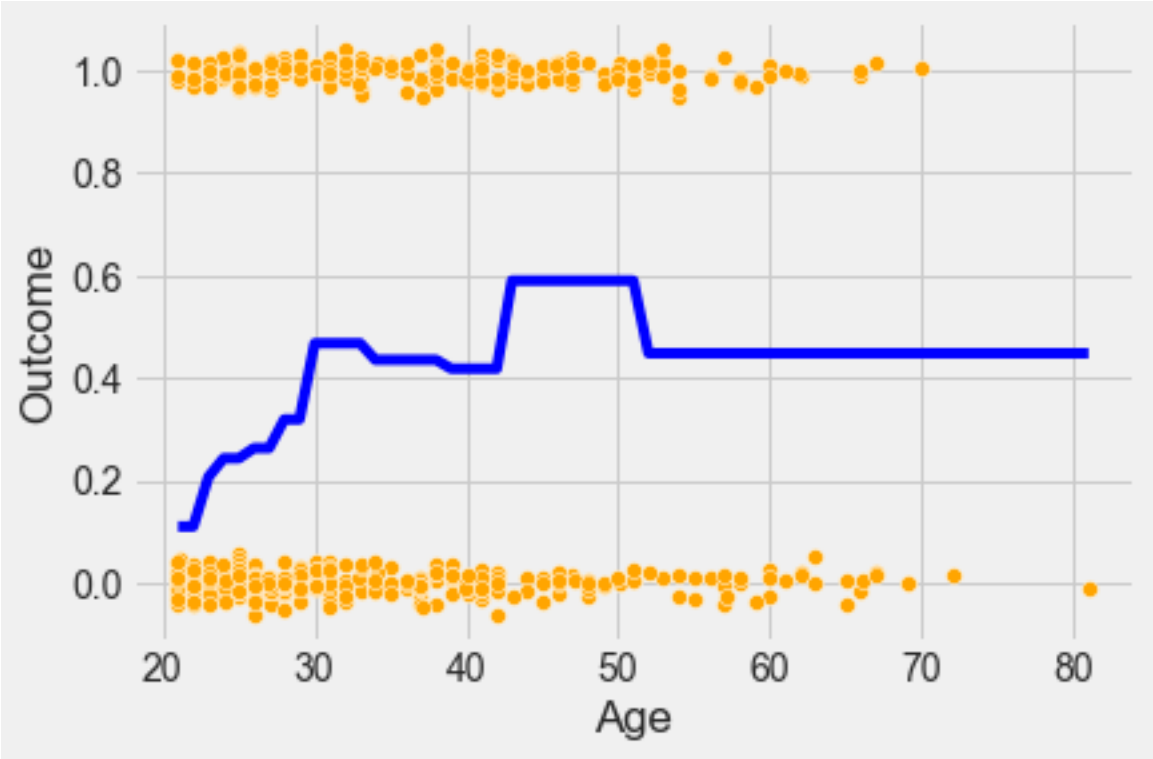
```
Text(0.5, 0, 'Age_bin')
```




```
#Model with binned Age
def jitter(values,j):
    return values + np.random.normal(j,0.02,values.shape)
sns.scatterplot(x = jitter(train.Age,0), y = jitter(train.Outcome,0), data = train, color
logit_model = sm.logit(formula = 'Outcome~' + '+'.join(['age'+str(x) for x in range(1,10)]
sns.lineplot(x = 'Age', y= logit_model.predict(train), data = train, color = 'blue')
```

```
Optimization terminated successfully.
         Current function value: 0.585956
         Iterations 6
```

```
<AxesSubplot:xlabel='Age', ylabel='Outcome'>
```




```
logit_model.summary()
```

Table 8.14: Logit Regression Results

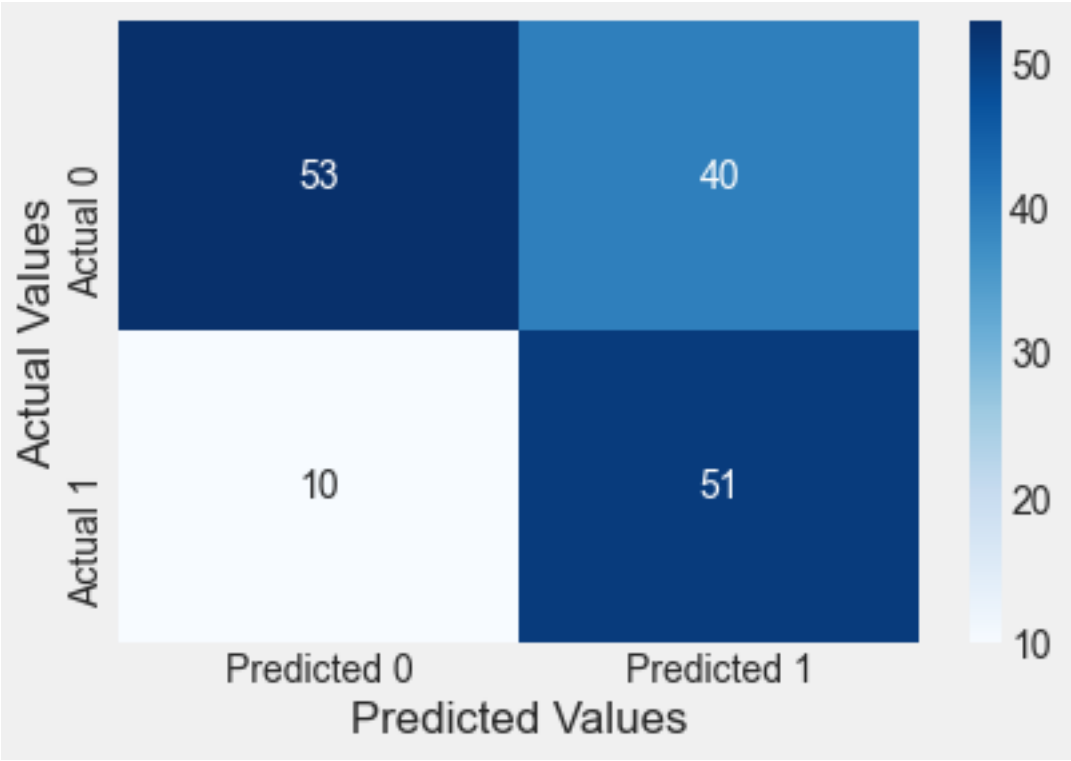
Dep. Variable:	Outcome	No. Observations:	614
Model:	Logit	Df Residuals:	604
Method:	MLE	Df Model:	9
Date:	Sun, 19 Feb 2023	Pseudo R-squ.:	0.08318
Time:	14:19:51	Log-Likelihood:	-359.78
converged:	True	LL-Null:	-392.42
Covariance Type:	nonrobust	LLR p-value:	1.273e-10

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-2.0898	0.306	-6.829	0.000	-2.690	-1.490
age1	0.7461	0.551	1.354	0.176	-0.334	1.826
age2	0.9548	0.409	2.336	0.019	0.154	1.756
age3	1.0602	0.429	2.471	0.013	0.219	1.901
age4	1.3321	0.438	3.044	0.002	0.474	2.190
age5	1.9606	0.398	4.926	0.000	1.180	2.741
age6	1.8303	0.399	4.586	0.000	1.048	2.612
age7	1.7596	0.410	4.288	0.000	0.955	2.564
age8	2.4544	0.402	6.109	0.000	1.667	3.242
age9	1.8822	0.404	4.657	0.000	1.090	2.674

Note that the probability of having diabetes for each age bin is a constant, as per the above plot.

```
confusion_matrix_test(test,test.Outcome,logit_model,0.3)
```

```
Classification accuracy = 67.5%
```



Binning `Age` provides a similar result as compared to the model with the quadratic transformation of `Age`.

```
train.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	A
158	2	88	74	19	53	29.0	0.229	22
251	2	129	84	0	0	28.0	0.284	27
631	0	102	78	40	90	34.5	0.238	24
757	0	123	72	0	0	36.3	0.258	52
689	1	144	82	46	180	46.1	0.335	46

```
#Model with the quadratic transformation of Age and more predictors
logit_model_diabetes = sm.logit(formula = 'Outcome~Age+I(Age**2)+Glucose+BloodPressure+BMI
logit_model_diabetes.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.470478
         Iterations 6
```

Table 8.17: Logit Regression Results

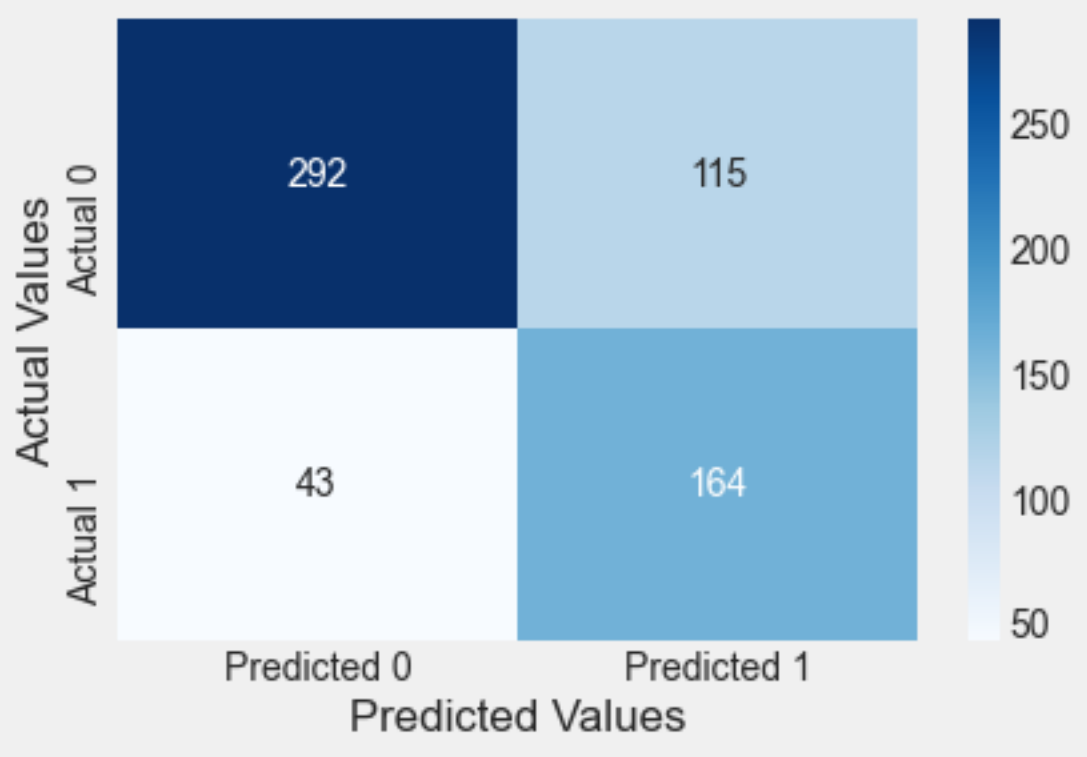
Dep. Variable:	Outcome	No. Observations:	614
Model:	Logit	Df Residuals:	607
Method:	MLE	Df Model:	6
Date:	Thu, 23 Feb 2023	Pseudo R-squ.:	0.2639
Time:	10:26:00	Log-Likelihood:	-288.87
converged:	True	LL-Null:	-392.42
Covariance Type:	nonrobust	LLR p-value:	5.878e-42

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-12.3347	1.282	-9.621	0.000	-14.847	-9.822
Age	0.2852	0.056	5.121	0.000	0.176	0.394
I(Age ** 2)	-0.0030	0.001	-4.453	0.000	-0.004	-0.002
Glucose	0.0309	0.004	8.199	0.000	0.024	0.038
BloodPressure	-0.0141	0.006	-2.426	0.015	-0.025	-0.003
BMI	0.0800	0.016	4.978	0.000	0.049	0.112
DiabetesPedigreeFunction	0.7138	0.322	2.213	0.027	0.082	1.346

Adding more predictors has increased the log likelihood of the model as expected.

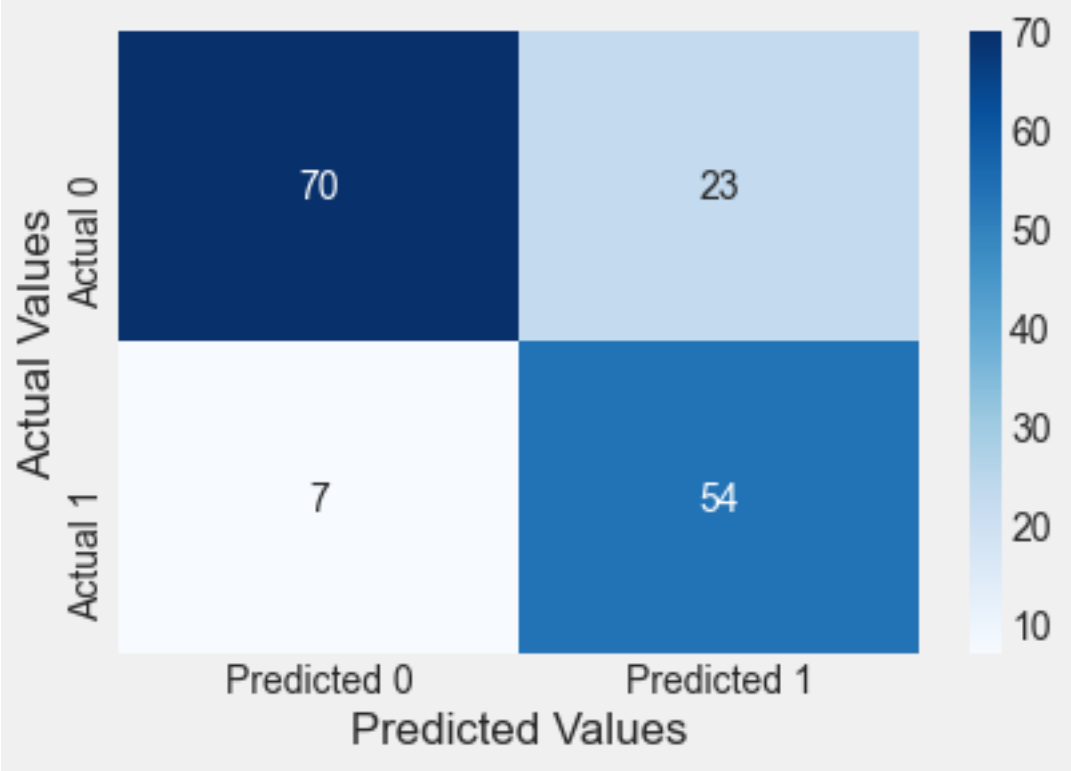
```
confusion_matrix_train(logit_model_diabetes,cutoff=0.3)
```

```
Classification accuracy = 74.3%
```




```
confusion_matrix_test(test,test.Outcome,logit_model_diabetes,0.3)
```

```
Classification accuracy = 80.5%
```



The model with more predictors also has lesser number of *false negatives*, and higher overall classification accuracy.

How many bins must you make for `Age` to get the most accurate model?
If the number of bins are too less, the trend may not be captured accurately. If the number of bins are too many, it may lead to overfitting of the model. There is an optimal value of the number of bins that captures the trend, but does not overfit. A couple of ways of estimating the optimal number of bins can be:

1. The number of bins for which the trend continues to be "almost" the same for several samples of the data.
2. Testing the model on multiple test datasets.

Optimizing the number of bins for each predictor may be a time-consuming exercises. You may do it for your course project. However, we will not do it here in the class notes.

8.6 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`****).*

8.6.1 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.

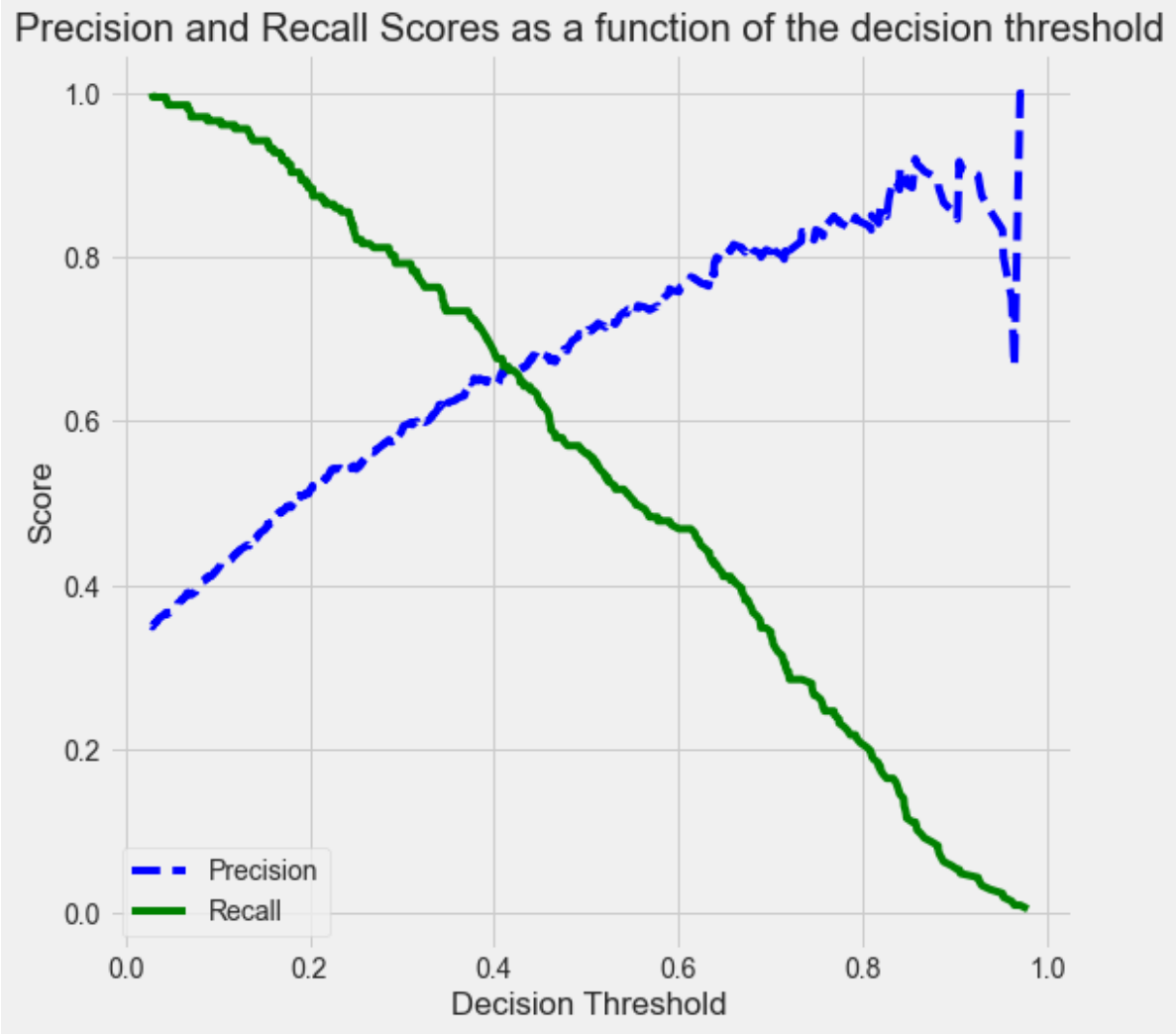
$$\text{recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

==> `88.52%`

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

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

```
from sklearn.metrics import precision_recall_curve
y=train.Outcome
ypred = logit_model_diabetes.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`.

8.6.2 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.

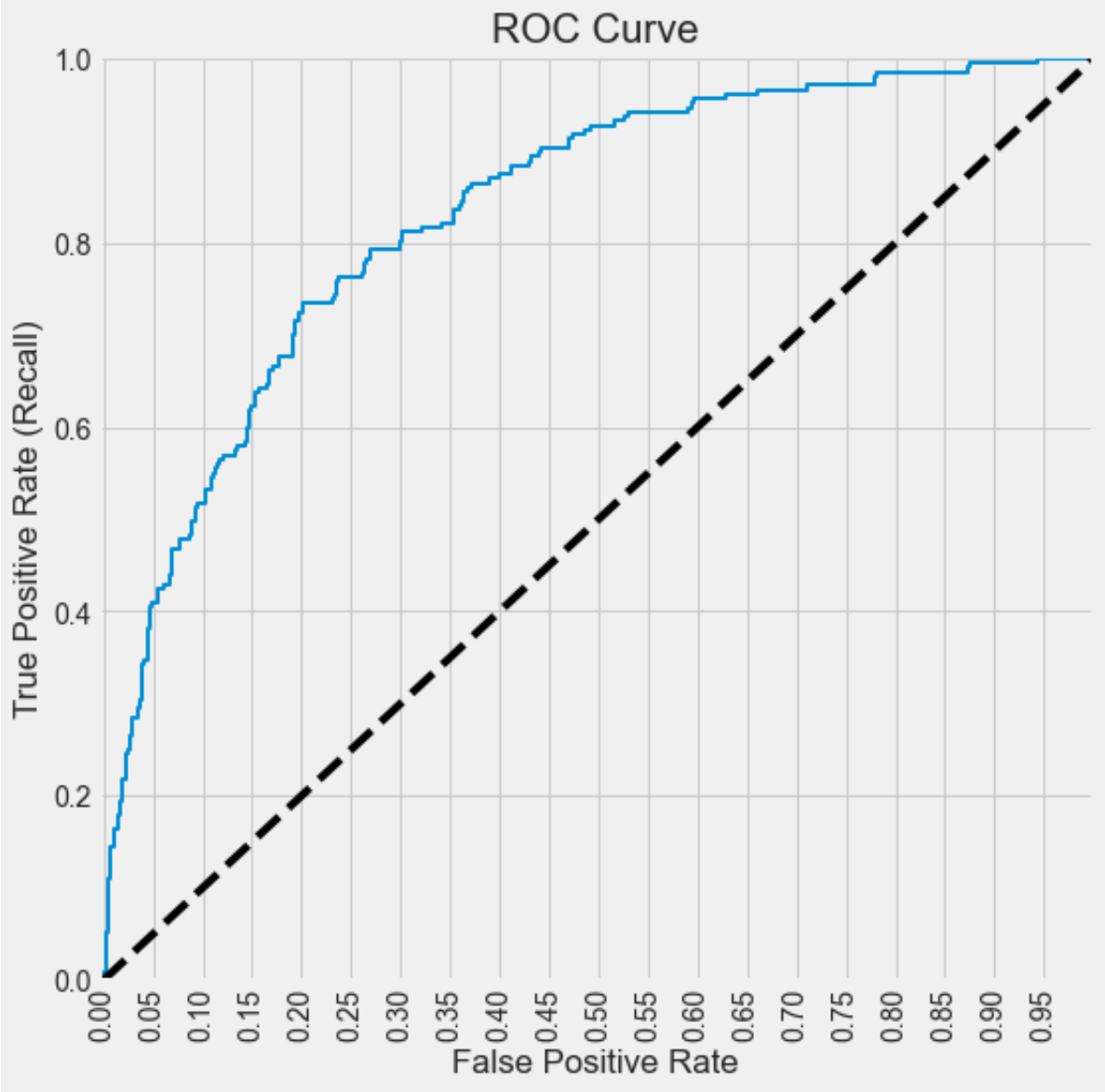
Here is good post by google developers on interpreting ROC-AUC, and its advantages / disadvantages.

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

```
from sklearn.metrics import roc_curve, auc
y=train.Outcome
ypred = logit_model_diabetes.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)
```

```
0.8325914847653979
```



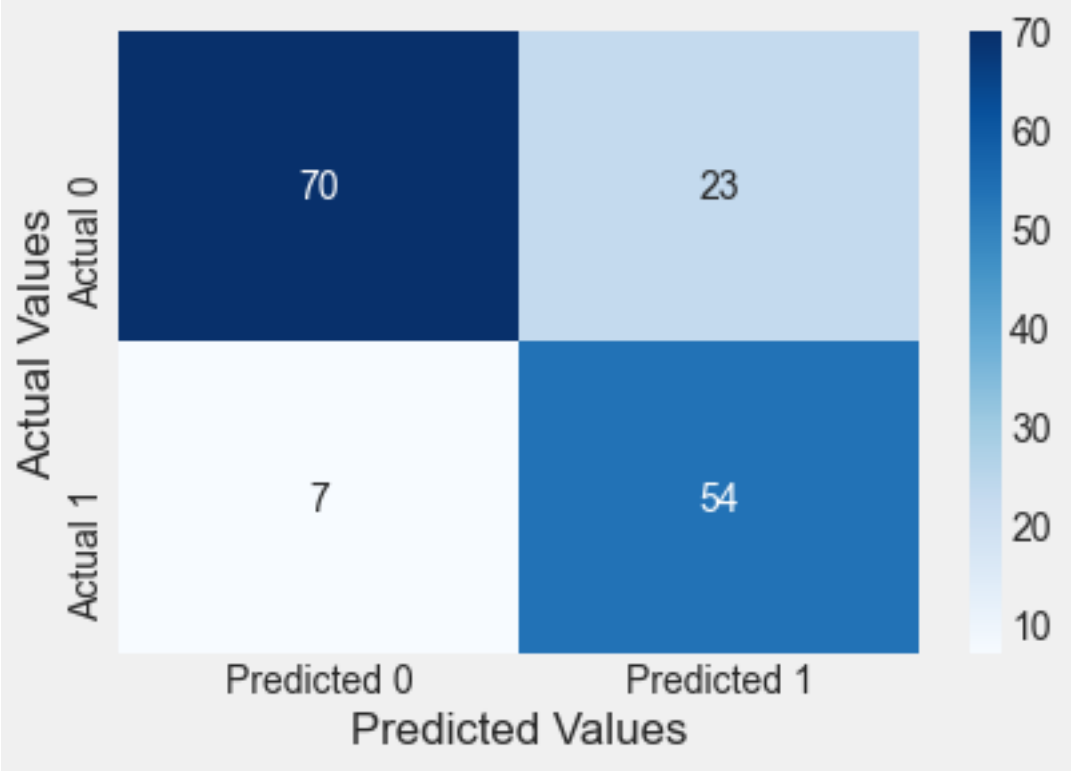
Q: How are the values of the `auc_thresholds` chosen to make the ROC curve? Why does it look like a step function?

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.Outcome,logit_model_diabetes,0.3)
```

```
Classification accuracy = 80.5%
Precision = 70.1%
TPR or Recall = 88.5%
FNR = 11.5%
FPR = 24.7%
ROC-AUC = 90.1%
```


70
23
7
54
Actual Values
Actual 0
Actual 1
Predicted 0
Predicted 1
Predicted Values
70
60
50
40
30
20
10

Part III

Variable selection & Regularization

9 Best subset and Stepwise selection

Read section 6.1 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.

9.1 Best subsets selection

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

```
trainf = pd.read_csv('./Datasets/house_feature_train.csv')
trainp = pd.read_csv('./Datasets/house_price_train.csv')
testf = pd.read_csv('./Datasets/house_feature_test.csv')
testp = pd.read_csv('./Datasets/house_price_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	house_id	house_age	distance_MRT	number_convenience_stores	latitude	longitude	house_pri
0	210	5.2	390.5684	5	24.97937	121.54245	2724.84
1	190	35.3	616.5735	8	24.97945	121.53642	1789.29
2	328	15.9	1497.7130	3	24.97003	121.51696	556.96
3	5	7.1	2175.0300	3	24.96305	121.51254	1030.41
4	412	8.1	104.8101	5	24.96674	121.54067	2756.25

Develop a model to predict house price using the rest of the columns as predictors (except `house_id`).

```
#Model with log house price as the response and the remaining variables as predictors
model = sm.ols('np.log(house_price)~house_age+distance_MRT+number_convenience_stores+latit
longitude', data = train).fit()
model.summary()
```

Table 9.2: OLS Regression Results

Dep. Variable:	np.log(house_price)	R-squared:	0.772
Model:	OLS	Adj. R-squared:	0.767
Method:	Least Squares	F-statistic:	181.8
Date:	Thu, 16 Feb 2023	Prob (F-statistic):	4.47e-84
Time:	18:31:07	Log-Likelihood:	-118.47
No. Observations:	275	AIC:	248.9
Df Residuals:	269	BIC:	270.6
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-482.9401	312.000	-1.548	0.123	-1097.212	131.332
house_age	-0.0131	0.002	-6.437	0.000	-0.017	-0.009
distance_MRT	-0.0003	3.69e-05	-8.318	0.000	-0.000	-0.000
number_convenience_stores	0.0598	0.010	6.247	0.000	0.041	0.079
latitude	18.7044	2.353	7.951	0.000	14.073	23.336
longitude	0.1923	2.465	0.078	0.938	-4.660	5.045

Omnibus:	4.413	Durbin-Watson:	2.260
Prob(Omnibus):	0.110	Jarque-Bera (JB):	5.515
Skew:	0.077	Prob(JB):	0.0634
Kurtosis:	3.677	Cond. No.	2.28e+07

Find the best subset of predictors that can predict house price in a linear regression model.

```
#Creating a set of predictors from which we need to find the best subset of predictors
X = train[['house_age','number_convenience_stores','latitude', 'longitude','distance_MRT']
```

9.1.1 Best subset selection algorithm

Now, we will implement the algorithm of finding the best subset of predictors from amongst all sets of predictors.

```
#Function to develop a model based on all predictors in predictor_subset
def processSubset(predictor_subset):
    # Fit model on feature_set and calculate R-squared
    model = sm.ols('np.log(house_price)~' + '+'.join(predictor_subset),data = train).fit()
    Rsquared = model.rsquared
    return {"model":model, "Rsquared":Rsquared}
```

```
#Function to select the best model amongst all models with 'k' predictors
def getBest_model(k):
    tic = time.time()
    results = []
    for combo in itertools.combinations(X.columns, k):
        results.append(processSubset((list(combo))))

    # Wrap everything up in a dataframe
    models = pd.DataFrame(results)

    # Choose the model with the highest RSS
    best_model = models.loc[models['Rsquared'].argmax()]

    toc = time.time()
    print("Processed", models.shape[0], "models on", k, "predictors in", (toc-tic), "secon
    return best_model
```

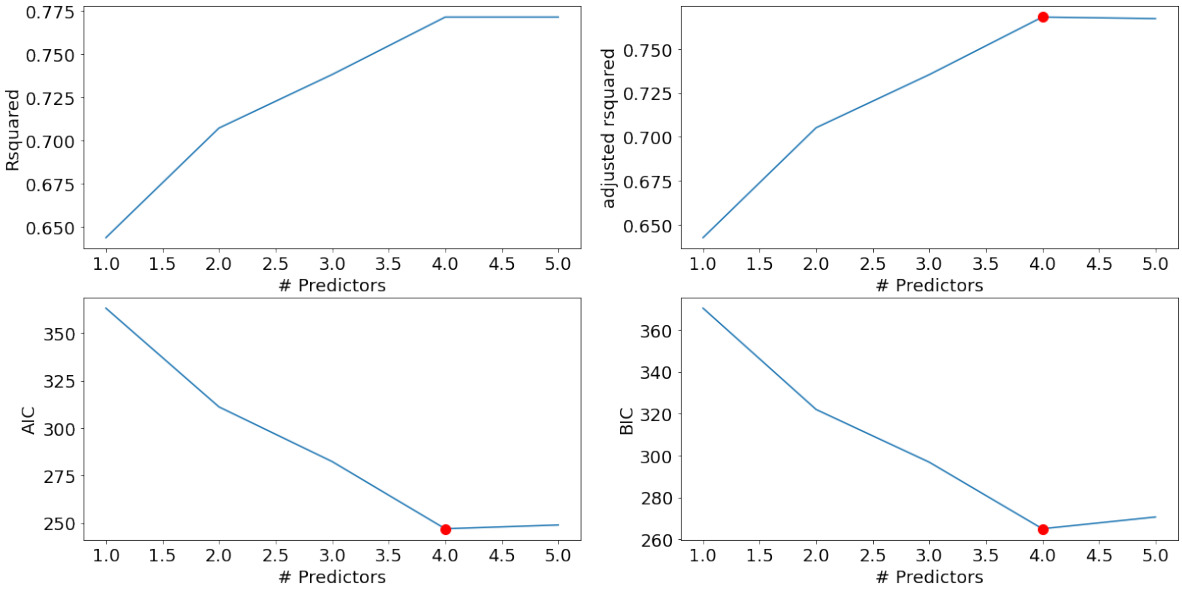
```
#Function to select the best model amongst the best models for 'k' predictors, where k = 1
models_best = pd.DataFrame(columns=["Rsquared", "model"])

tic = time.time()
for i in range(1,1+X.shape[1]):
    models_best.loc[i] = getBest_model(i)

toc = time.time()
print("Total elapsed time:", (toc-tic), "seconds.")
```

```
Processed 5 models on 1 predictors in 0.02393651008605957 seconds.
Processed 10 models on 2 predictors in 0.04688239097595215 seconds.
```

```
Processed 10 models on 3 predictors in 0.04986691474914551 seconds.
Processed 5 models on 4 predictors in 0.029920578002929688 seconds.
Processed 1 models on 5 predictors in 0.008975982666015625 seconds.
Total elapsed time: 0.17253828048706055 seconds.
```

```
def best_sub_plots():
    plt.figure(figsize=(20,10))
    plt.rcParams.update({'font.size': 18, 'lines.markersize': 10})

    # Set up a 2x2 grid so we can look at 4 plots at once
    plt.subplot(2, 2, 1)

    # We will now plot a red dot to indicate the model with the largest adjusted R^2 stati
    # The argmax() function can be used to identify the location of the maximum point of a
    plt.plot(models_best["Rsquared"])
    plt.xlabel('# Predictors')
    plt.ylabel('Rsquared')

    # We will now plot a red dot to indicate the model with the largest adjusted R^2 stati
    # The argmax() function can be used to identify the location of the maximum point of a

    rsquared_adj = models_best.apply(lambda row: row[1].rsquared_adj, axis=1)

    plt.subplot(2, 2, 2)
    plt.plot(rsquared_adj)
    plt.plot(1+rsquared_adj.argmax(), rsquared_adj.max(), "or")
    plt.xlabel('# Predictors')
    plt.ylabel('adjusted rsquared')

    # We'll do the same for AIC and BIC, this time looking for the models with the SMALLES
    aic = models_best.apply(lambda row: row[1].aic, axis=1)

    plt.subplot(2, 2, 3)
    plt.plot(aic)
    plt.plot(1+aic.argmin(), aic.min(), "or")
    plt.xlabel('# Predictors')
    plt.ylabel('AIC')

    bic = models_best.apply(lambda row: row[1].bic, axis=1)

    plt.subplot(2, 2, 4)
```

```
    plt.plot(bic)
    plt.plot(1+bic.argmin(), bic.min(), "or")
    plt.xlabel('# Predictors')
    plt.ylabel('BIC')
best_sub_plots()
```



The model with 4 predictors is the best model, according to all 3 criteria - Adjusted R-squared, AIC and BIC.

Note that we have not considered the null model (i.e., the model with only the intercept and no predictors) explicitly in the best subsets algorithm. However, the null model is considered when selecting the best model. The R-squared and the adjusted R-squared for the null model is 0. So, if the adjusted R-squared of all the models with at least one predictor is negative, then the null model will be the best model.

```
best_subset_model = models_best.loc[4,'model']
models_best.loc[4,'model'].summary()
```

Table 9.5: OLS Regression Results

Dep. Variable:	np.log(house_price)	R-squared:	0.772
Model:	OLS	Adj. R-squared:	0.768
Method:	Least Squares	F-statistic:	228.0
Date:	Thu, 16 Feb 2023	Prob (F-statistic):	2.79e-85

Time:	19:51:50	Log-Likelihood:	-118.47
No. Observations:	275	AIC:	246.9
Df Residuals:	270	BIC:	265.0
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-459.0262	58.231	-7.883	0.000	-573.671	-344.381
house_age	-0.0131	0.002	-6.451	0.000	-0.017	-0.009
number_convenience_stores	0.0597	0.010	6.271	0.000	0.041	0.078
latitude	18.6828	2.332	8.012	0.000	14.092	23.274
distance_MRT	-0.0003	2.53e-05	-12.221	0.000	-0.000	-0.000

Omnibus:	4.422	Durbin-Watson:	2.261
Prob(Omnibus):	0.110	Jarque-Bera (JB):	5.555
Skew:	0.073	Prob(JB):	0.0622
Kurtosis:	3.681	Cond. No.	4.25e+06

```
#Finding the RMSE of the model selected using the best subset selection procedure
pred_price = np.exp(best_subset_model.predict(test))
np.sqrt(((pred_price - test.house_price)**2).mean())
```

```
403.4635674362065
```

```
#RMSE of the model using all the predictors
model = sm.ols('np.log(house_price)~' + '+'.join(X.columns),data = train).fit()
pred_price = np.exp(model.predict(test))
np.sqrt(((pred_price - test.house_price)**2).mean())
```

```
403.8409399214197
```

The RMSE of the best subset model is similar to the RMSE of the model with all the predictors. This is because longitude varies only in [121.47, 121.57]. The coefficient of longitude is 0.1923 in the model with all the predictors. So, the change in the response due to longitude is in [23.36, 23.38]. This change in the response due to longitude is almost a constant, and hence is adjusted in the intercept of the model without longitude. Note the intercept of the model without longitude is 23.91 more than the intercept of the model with longitude.

```
[0.1923*train.longitude.min(),0.1923*train.longitude.max()]
```

```
[23.359359818999998, 23.377193721]
```

9.1.2 Including interactions for best subset selection

Let's perform best subset selection including all the predictors and their 2-factor interactions

```
#Creating a dataframe with all the predictors
X = train[['house_age', 'distance_MRT', 'number_convenience_stores','latitude','longitude'
#Since 'X' will change when we include interactions, we need a backup containing all indiv
X_backup = train[['house_age', 'distance_MRT', 'number_convenience_stores','latitude','lon
```

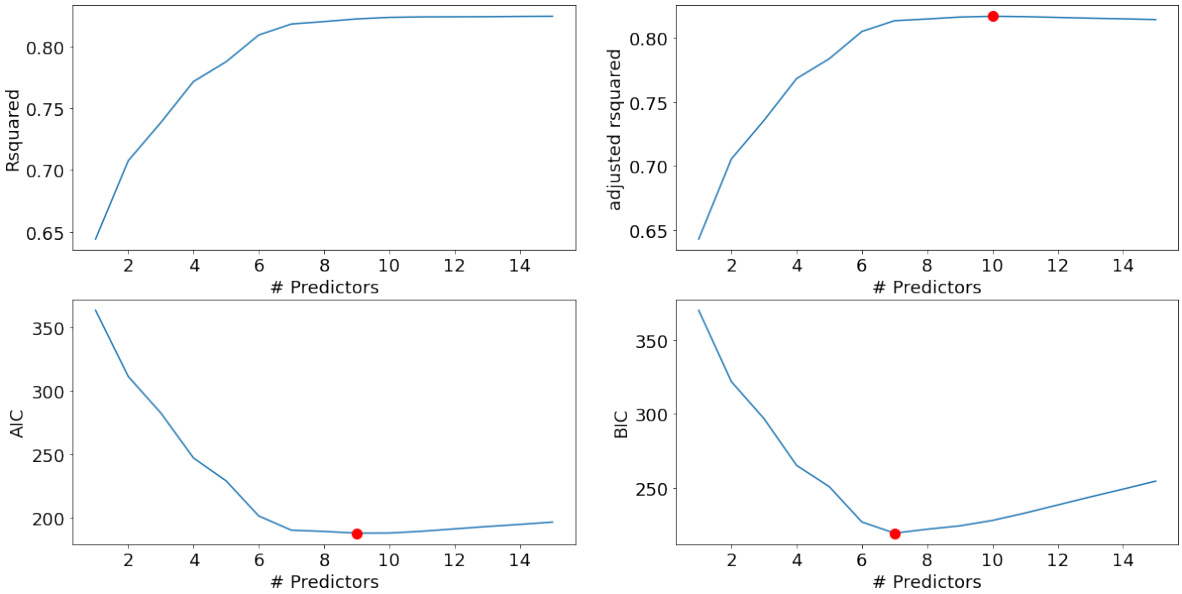
```
#Including 2-factor interactions of predictors in train and 'X'. Note that we need train t
#find 'k' variable subsets from amongst all the predictors under consideration
for combo in itertools.combinations(X_backup.columns, 2):
    train['_'.join(combo)] = train[combo[0]]*train[combo[1]]
    test['_'.join(combo)] = test[combo[0]]*test[combo[1]]
    X.loc[:,'_'.join(combo)] = train.loc[:,'_'.join(combo)]
```

```
models_best = pd.DataFrame(columns=["Rsquared", "model"])

tic = time.time()
for i in range(1,1+X.shape[1]):
    models_best.loc[i] = getBest_model(i)

toc = time.time()
print("Total elapsed time:", (toc-tic), "seconds.")
```

```
Processed 15 models on 1 predictors in 0.07200050354003906 seconds.
Processed 105 models on 2 predictors in 0.536522388458252 seconds.
Processed 455 models on 3 predictors in 2.6639997959136963 seconds.
Processed 1365 models on 4 predictors in 9.176022052764893 seconds.
Processed 3003 models on 5 predictors in 24.184194803237915 seconds.
Processed 5005 models on 6 predictors in 43.54697918891907 seconds.
Processed 6435 models on 7 predictors in 65.83688187599182 seconds.
Processed 6435 models on 8 predictors in 78.97277760505676 seconds.
Processed 5005 models on 9 predictors in 64.53991365432739 seconds.
```

```
Processed 3003 models on 10 predictors in 38.39328980445862 seconds.
Processed 1365 models on 11 predictors in 18.715795755386353 seconds.
Processed 455 models on 12 predictors in 6.93279504776001 seconds.
Processed 105 models on 13 predictors in 1.6240253448486328 seconds.
Processed 15 models on 14 predictors in 0.256000280380249 seconds.
Processed 1 models on 15 predictors in 0.024001121520996094 seconds.
Total elapsed time: 356.2638840675354 seconds.
```

```
best_sub_plots()
```



The model with 7 predictors is the best model based on the BIC criterion, and very close to the best model based on the AIC and Adjusted R-squared criteria. Let us select the model with 7 predictors.

```
best_interaction_model = models_best['model'][7]
best_interaction_model.summary()
```

Table 9.8: OLS Regression Results

Dep. Variable:	np.log(house_price)	R-squared:	0.818
Model:	OLS	Adj. R-squared:	0.814
Method:	Least Squares	F-statistic:	171.7
Date:	Thu, 16 Feb 2023	Prob (F-statistic):	5.29e-95

Time:	20:17:02	Log-Likelihood:	-87.046
No. Observations:	275	AIC:	190.1
Df Residuals:	267	BIC:	219.0
Df Model:	7		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-1316.6156	135.152	-9.742	0.000	-1582.716	-1050.515
distance_MRT	0.2424	0.044	5.539	0.000	0.156	0.329
number_convenience_stores	152.0179	23.356	6.509	0.000	106.033	198.003
latitude	53.0284	5.413	9.797	0.000	42.371	63.686
house_age_longitude	-0.0001	1.51e-05	-6.842	0.000	-0.000	-7.36e-05
distance_MRT_number_convenience_stores	-5.691e-05	1.19e-05	-4.763	0.000	-8.04e-05	-3.34e-05
distance_MRT_latitude	-0.0097	0.002	-5.544	0.000	-0.013	-0.006
number_convenience_stores_latitude	-6.0847	0.935	-6.506	0.000	-7.926	-4.243

Omnibus:	5.350	Durbin-Watson:	2.136
Prob(Omnibus):	0.069	Jarque-Bera (JB):	7.524
Skew:	0.045	Prob(JB):	0.0232
Kurtosis:	3.805	Cond. No.	2.78e+08

Note that only 3 of the 10 two factor interactions are included in the best subset model, and the predictor `longitude` has been dropped.

```
#Finding the RMSE of the model selected using the best subset selection procedure, where t
#include 2-factor interactions
pred_price = np.exp(best_interaction_model.predict(test))
np.sqrt(((pred_price - test.house_price)**2).mean())
```

```
346.4100962681362
```

```
#Model with the predictors and all their 2-factor interactions
model = sm.ols('np.log(house_price)~' + '+'.join(X.columns),data = train).fit()
model.summary()
```

Table 9.11: OLS Regression Results

Dep. Variable:	np.log(house_price)	R-squared:	0.825
Model:	OLS	Adj. R-squared:	0.814
Method:	Least Squares	F-statistic:	81.14
Date:	Thu, 16 Feb 2023	Prob (F-statistic):	1.33e-88
Time:	20:13:01	Log-Likelihood:	-82.228
No. Observations:	275	AIC:	196.5
Df Residuals:	259	BIC:	254.3
Df Model:	15		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	7.455e+05	1.03e+06	0.725	0.469	-1.28e+06	2.77e+06
house_age	83.1021	40.562	2.049	0.041	3.228	162.976
distance_MRT	0.1391	0.174	0.798	0.425	-0.204	0.482
number_convenience_stores	252.5261	212.276	1.190	0.235	-165.481	670.533
latitude	-2.992e+04	4.12e+04	-0.727	0.468	-1.11e+05	5.12e+04
longitude	-6144.1732	8454.331	-0.727	0.468	-2.28e+04	1.05e+04
house_age_distance_MRT	-2.904e-06	4.44e-06	-0.654	0.514	-1.16e-05	5.84e-06
house_age_number_convenience_stores	0.0011	0.001	1.409	0.160	-0.000	0.003
house_age_latitude	0.2119	0.261	0.811	0.418	-0.303	0.726
house_age_longitude	-0.7274	0.330	-2.207	0.028	-1.376	-0.078
distance_MRT_number_convenience_stores	-6.192e-05	1.99e-05	-3.115	0.002	-0.000	-2.28e-05
distance_MRT_latitude	-0.0082	0.003	-2.387	0.018	-0.015	-0.001
distance_MRT_longitude	0.0005	0.001	0.417	0.677	-0.002	0.003
number_convenience_stores_latitude	-6.4014	1.113	-5.753	0.000	-8.592	-4.210
number_convenience_stores_longitude	-0.7620	1.700	-0.448	0.654	-4.109	2.585
latitude_longitude	246.5995	338.773	0.728	0.467	-420.500	913.699

Omnibus:	3.911	Durbin-Watson:	2.134
Prob(Omnibus):	0.142	Jarque-Bera (JB):	4.552
Skew:	0.090	Prob(JB):	0.103
Kurtosis:	3.604	Cond. No.	1.05e+13

```
# RMSE of the model using all the predictors and their 2-factor interactions
pred_price = np.exp(model.predict(test))
np.sqrt(((pred_price - test.house_price)**2).mean())
```

```
360.40099598821615
```

The best subset model seems to be slightly better than the model with all the predictors, based on the RMSE on test data.

9.2 Stepwise selection

Best subset selection cannot be used in case of even a slightly large number of predictors. In the previous example, we had 15 predictors. The number of models that we developed to find the best subset of predictors from the set of 15 predictors was $2^{15} \approx 32,000$. In case of 20 predictors, the number of models to use the best subset selection approach will be $2^{20} \approx 1$ million, which is computationally too expensive. Due to this limitation of the best subsets selection method, we will use stepwise regression, which explores a far more restricted set of models, and thus is an attractive alternative to the best subset selection method.

9.3 Forward stepwise selection

Source - Page 229: "Forward stepwise selection is a computationally efficient alternative to best subset selection. While the best subset selection procedure considers all 2^p possible models containing subsets of the p predictors, forward stepwise considers a much smaller set of models. Forward stepwise selection begins with a model containing no predictors, and then adds predictors to the model, one-at-a-time, until all of the predictors are in the model. In particular, at each step the variable that gives the greatest additional improvement to the fit is added to the model."

```
#Function to find the best predictor out of p-k predictors and add it to the model contain
def forward(predictors):

    # Pull out predictors we still need to process
    remaining_predictors = [p for p in X.columns if p not in predictors]

    tic = time.time()

    results = []

    for p in remaining_predictors:
        results.append(processSubset(predictors+[p]))

    # Wrap everything up in a nice dataframe
```

```
    models = pd.DataFrame(results)

    # Choose the model with the highest RSS
    best_model = models.loc[models['Rsquared'].argmax()]

    toc = time.time()
    print("Processed ", models.shape[0], "models on", len(predictors)+1, "predictors in",

    # Return the best model, along with some other useful information about the model
    return best_model
```

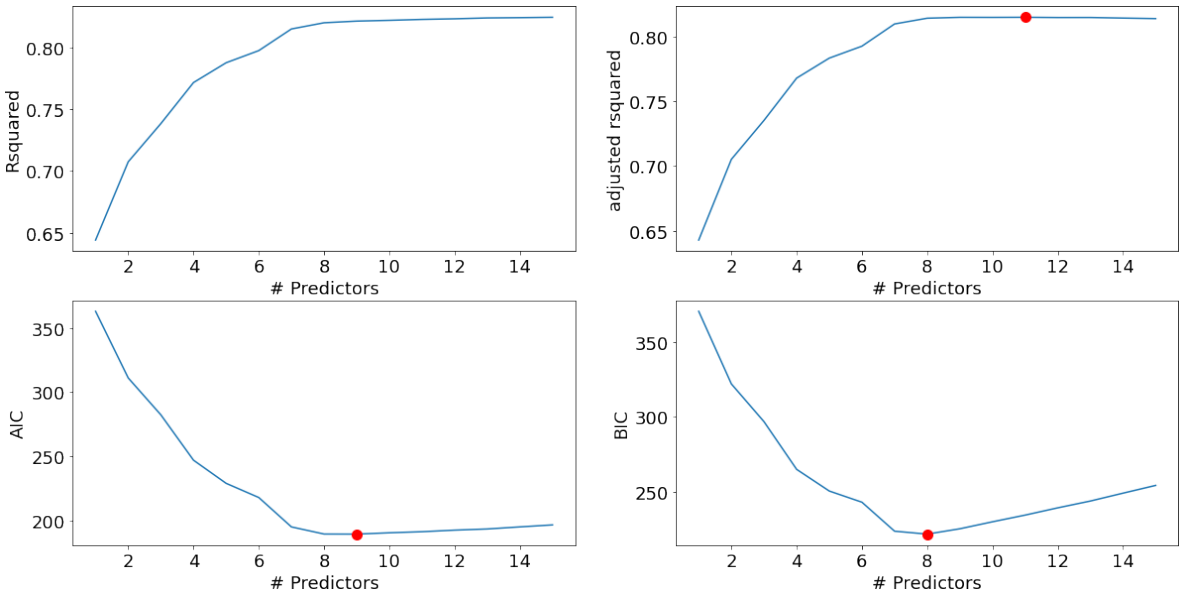
```
def forward_selection():
    models_best = pd.DataFrame(columns=["Rsquared", "model"])

    tic = time.time()
    predictors = []

    for i in range(1,len(X.columns)+1):
        models_best.loc[i] = forward(predictors)
        predictors = list(models_best.loc[i]["model"].params.index[1:])

    toc = time.time()
    print("Total elapsed time:", (toc-tic), "seconds.")
    return models_best
```

```
models_best = forward_selection()
```

```
Processed  15 models on 1 predictors in 0.06280803680419922 seconds.
Processed  14 models on 2 predictors in 0.054885149002075195 seconds.
Processed  13 models on 3 predictors in 0.05983686447143555 seconds.
Processed  12 models on 4 predictors in 0.06781768798828125 seconds.
Processed  11 models on 5 predictors in 0.07380270957946777 seconds.
Processed  10 models on 6 predictors in 0.07380390167236328 seconds.
Processed  9 models on 7 predictors in 0.06981182098388672 seconds.
Processed  8 models on 8 predictors in 0.07480072975158691 seconds.
Processed  7 models on 9 predictors in 0.0718071460723877 seconds.
Processed  6 models on 10 predictors in 0.06380081176757812 seconds.
Processed  5 models on 11 predictors in 0.054854631423950195 seconds.
Processed  4 models on 12 predictors in 0.05385565757751465 seconds.
Processed  3 models on 13 predictors in 0.04188799858093262 seconds.
Processed  2 models on 14 predictors in 0.027925491333007812 seconds.
```

```
Processed  1 models on 15 predictors in 0.016956090927124023 seconds.
Total elapsed time: 0.9055600166320801 seconds.
```

```
best_sub_plots()
```



The model with 8 predictors is the best model based on the BIC criterion, and very close to the best model based on the AIC and Adjusted R-squared criteria. Let us select the model with 8 predictors.

```
best_fwd_reg_model = models_best['model'][8]
best_fwd_reg_model.summary()
```

Table 9.14: OLS Regression Results

Dep. Variable:	np.log(house_price)	R-squared:	0.820
Model:	OLS	Adj. R-squared:	0.815
Method:	Least Squares	F-statistic:	151.6
Date:	Thu, 16 Feb 2023	Prob (F-statistic):	1.91e-94
Time:	20:35:14	Log-Likelihood:	-85.667
No. Observations:	275	AIC:	189.3
Df Residuals:	266	BIC:	221.9
Df Model:	8		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-1365.5045	154.113	-8.860	0.000	-1668.942	-1062.067
distance_MRT_longitude	0.0021	0.000	5.062	0.000	0.001	0.003
latitude	54.9844	6.171	8.909	0.000	42.833	67.136
house_age_longitude	-0.3240	0.119	-2.725	0.007	-0.558	-0.090
number_convenience_stores_longitude	1.3242	0.212	6.246	0.000	0.907	1.742
distance_MRT_number_convenience_stores	-4.805e-05	1.21e-05	-3.973	0.000	-7.19e-05	-2.42e-05
number_convenience_stores_latitude	-6.4419	1.032	-6.243	0.000	-8.473	-4.410
distance_MRT_latitude	-0.0101	0.002	-5.067	0.000	-0.014	-0.006
house_age	39.3625	14.450	2.724	0.007	10.911	67.814

Omnibus:	5.017	Durbin-Watson:	2.176
Prob(Omnibus):	0.081	Jarque-Bera (JB):	6.923
Skew:	0.022	Prob(JB):	0.0314
Kurtosis:	3.776	Cond. No.	1.56e+09

```
#Finding the RMSE of the model selected using the forward selection procedure, where the p
#include 2-factor interactions
pred_price = np.exp(best_fwd_reg_model.predict(test))
np.sqrt(((pred_price - test.house_price)**2).mean())
```

```
364.2004089481364
```

We get a different model than what we got with the best subsets selection method. However, we got it in 0.9 seconds, instead of 6 minutes taken by the best subset selection algorithm. Note that this model has a higher RMSE as compared to the model obtained with the best subset selection procedure, which is expected. However, the RMSE is even slightly higher than the model that includes all the two factor interactions. This may be due to the following reasons:

- This may be due to chance - the test data set may be biased.
- The stepwise variable selection algorithms are greedy algorithms, and certainly don't guarantee the best model, or even a model better than the one without variable selection. However, in general, they are likely to provide a better model than the base model that includes all the predictors, especially if there are several predictors that are not associated with the response.

- For metrics such as adjusted R-squared, the adjustment is not directly tied to the model being more accurate on test data. The adjustment only ensures that the adjusted R-squared increases if the added predictor sufficiently reduces the RSS (Residual sum of squares) on training data.
- AIC is an unbiased estimate of test error. However, AIC will have some variance as we are using sample data for training the model.

9.4 Backward Stepwise Selection

Source - Page 231: "Like forward stepwise selection, backward stepwise selection provides an efficient alternative to best subset selection. However, unlike forward stepwise selection, it begins with the full least squares model containing all p predictors, and then iteratively removes the least useful predictor, one-at-a-time."

Let us try the backward selection procedure on the model with 15 predictors - *house_age, distance_MRT, number_convenience_stores, latitude, longitude* and their 2-factor interactions.

```
def backward(predictors):

    tic = time.time()

    results = []

    for combo in itertools.combinations(predictors, len(predictors)-1):
        results.append(processSubset(combo))

    # Wrap everything up in a nice dataframe
    models = pd.DataFrame(results)

    # Choose the model with the highest RSS
    best_model = models.loc[models['Rsquared'].argmax()]

    toc = time.time()
    print("Processed ", models.shape[0], "models on", len(predictors)-1, "predictors in",

    # Return the best model, along with some other useful information about the model
    return best_model
```

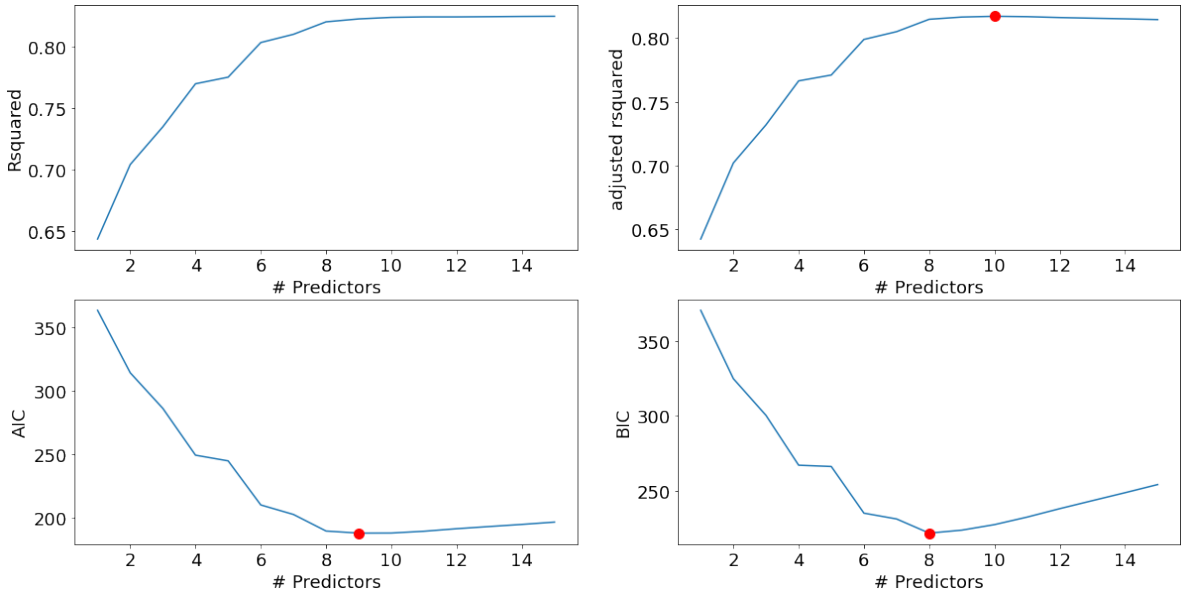
```
def backward_selection():
    models_best = pd.DataFrame(columns=["Rsquared", "model"], index = range(1,len(X.column
```

```
    tic = time.time()
    predictors = X.columns
    models_best.loc[len(predictors)] = processSubset(predictors)

    while(len(predictors) > 1):
        models_best.loc[len(predictors)-1] = backward(predictors)
        predictors = models_best.loc[len(predictors)-1]["model"].params.index[1:]

    toc = time.time()
    print("Total elapsed time:", (toc-tic), "seconds.")
    return models_best
```

```
models_best = backward_selection()
```

```
Processed  15 models on 14 predictors in 0.24733757972717285 seconds.
Processed  14 models on 13 predictors in 0.1765275001525879 seconds.
Processed  13 models on 12 predictors in 0.16356277465820312 seconds.
Processed  12 models on 11 predictors in 0.13364267349243164 seconds.
Processed  11 models on 10 predictors in 0.11968183517456055 seconds.
Processed  10 models on 9 predictors in 0.09571337699890137 seconds.
Processed  9 models on 8 predictors in 0.08377647399902344 seconds.
Processed  8 models on 7 predictors in 0.06981253623962402 seconds.
Processed  7 models on 6 predictors in 0.048902273178100586 seconds.
Processed  6 models on 5 predictors in 0.04088902473449707 seconds.
Processed  5 models on 4 predictors in 0.029920101165771484 seconds.
Processed  4 models on 3 predictors in 0.020944595336914062 seconds.
Processed  3 models on 2 predictors in 0.013962507247924805 seconds.
Processed  2 models on 1 predictors in 0.007978677749633789 seconds.
Total elapsed time: 1.286529779434204 seconds.
```

```
best_sub_plots()
```




```
best_bwd_reg_model = models_best['model'][8]
best_bwd_reg_model.summary()
```

Table 9.17: OLS Regression Results

Dep. Variable:	np.log(house_price)	R-squared:	0.820
Model:	OLS	Adj. R-squared:	0.815
Method:	Least Squares	F-statistic:	151.5
Date:	Thu, 16 Feb 2023	Prob (F-statistic):	2.00e-94
Time:	20:40:43	Log-Likelihood:	-85.714
No. Observations:	275	AIC:	189.4
Df Residuals:	266	BIC:	222.0
Df Model:	8		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-1317.5329	145.605	-9.049	0.000	-1604.218	-1030.847
house_age	57.3124	14.583	3.930	0.000	28.600	86.025
distance_MRT	0.2365	0.047	5.044	0.000	0.144	0.329
number_convenience_stores	154.8362	24.984	6.197	0.000	105.644	204.029
house_age_longitude	-0.4717	0.120	-3.931	0.000	-0.708	-0.235
distance_MRT_number_convenience_stores	-4.789e-05	1.24e-05	-3.869	0.000	-7.23e-05	-2.35e-05
distance_MRT_latitude	-0.0095	0.002	-5.050	0.000	-0.013	-0.006

number_convenience_stores_latitude	-6.1977	1.001	-6.194	0.000	-8.168	-4.228
latitude_longitude	0.4366	0.048	9.100	0.000	0.342	0.531

Omnibus:	4.945	Durbin-Watson:	2.137
Prob(Omnibus):	0.084	Jarque-Bera (JB):	6.228
Skew:	0.110	Prob(JB):	0.0444
Kurtosis:	3.703	Cond. No.	3.01e+08

We get a slightly different model than what we got with the best subsets selection method and the forward selection method. As in forward selection, we got it relatively very quickly (in 1.28 seconds), instead of 6 minutes taken by the best subset selection algorithm.

```
#Finding the RMSE of the model selected using the backward selection procedure, where the
#include 2-factor interactions
pred_price = np.exp(best_bwd_reg_model.predict(test))
np.sqrt(((pred_price - test.house_price)**2).mean())
```

```
363.63365786020694
```

Note that we have not considered the null model (i.e., the model with only the intercept and no predictors) explicitly in the forward and backward stepwise algorithms. However, the null model is considered when selecting the best model. The R-squared and the adjusted R-squared for the null model is 0. So, if the adjusted R-squared of all the models with at least one predictor is negative, then the null model will be the best model.

10 Ridge regression and Lasso

Read section 6.2 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.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge, RidgeCV, Lasso, LassoCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
```

```
trainf = pd.read_csv('./Datasets/house_feature_train.csv')
trainp = pd.read_csv('./Datasets/house_price_train.csv')
testf = pd.read_csv('./Datasets/house_feature_test.csv')
testp = pd.read_csv('./Datasets/house_price_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	house_id	house_age	distance_MRT	number_convenience_stores	latitude	longitude	house_pri
0	210	5.2	390.5684	5	24.97937	121.54245	2724.84
1	190	35.3	616.5735	8	24.97945	121.53642	1789.29
2	328	15.9	1497.7130	3	24.97003	121.51696	556.96
3	5	7.1	2175.0300	3	24.96305	121.51254	1030.41
4	412	8.1	104.8101	5	24.96674	121.54067	2756.25

10.1 Ridge regression

Let us develop a ridge regression model to predict house price based on the five house features.

```
#Taking the log transform of house_price as house prices have a right-skewed distribution
y = np.log(train.house_price)
```

10.1.1 Standardizing the predictors

```
#Standardizing predictors so that each of them have zero mean and unit variance

#Filtering all predictors
X = train.iloc[:,1:6];

#Defining a scaler object
scaler = StandardScaler()

#The scaler object will contain the mean and variance of each column (predictor) of X.
#These values will be useful to scale test data based on the same mean and variance as obt
scaler.fit(X)

#Using the scaler object (or the values of mean and variance stored in it) to standardize
Xstd = scaler.transform(X)
```

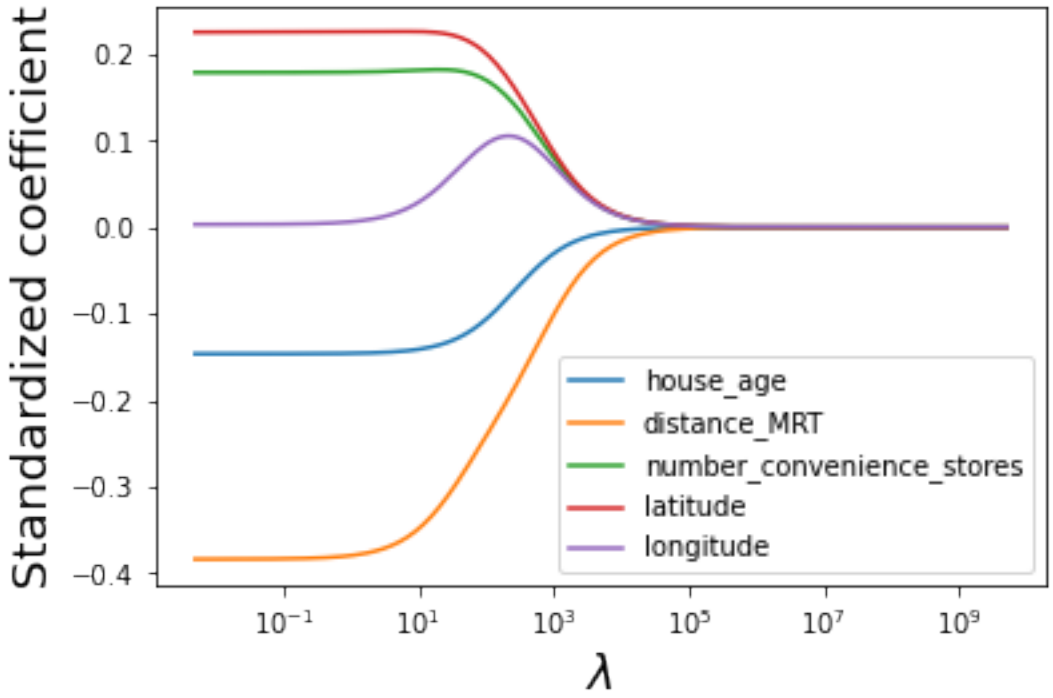
10.1.2 Optimizing the tuning parameter

```
#The tuning parameter lambda is referred as alpha in sklearn

#Creating a range of values of the tuning parameter to visualize the ridge regression coef
#for different values of the tuning parameter
alphas = 10**np.linspace(10,-2,200)*0.5
```

```
#Finding the ridge regression coefficients for increasing values of the tuning parameter
coefs = []
for a in alphas:
    ridge = Ridge(alpha = a)
    ridge.fit(Xstd, y)
```

```
    coefs.append(ridge.coef_)
```

```
#Visualizing the shrinkage in ridge regression coefficients with increasing values of the
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.plot(alphas, coefs)
plt.xscale('log')
plt.xlabel('$\lambda$')
plt.ylabel('Standardized coefficient')
plt.legend(train.columns[1:6]);
```



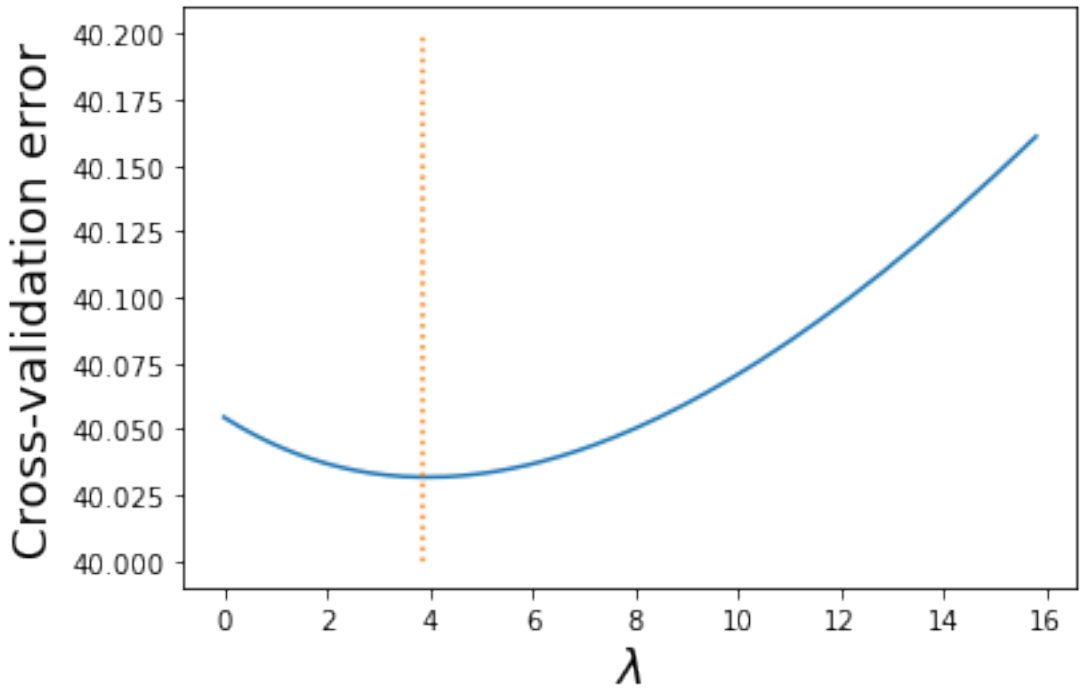

```
#Let us use cross validation to find the optimal value of the tuning parameter - lambda
#For the optimal lambda, the cross validation error will be the least

#Note that we are reducing the range of alpha so as to better visualize the minimum
alphas = 10**np.linspace(1.5,-3,200)*0.5
ridgecv = RidgeCV(alphas = alphas,store_cv_values=True)
ridgecv.fit(Xstd, y)
```

```
#Optimal value of the tuning parameter - lambda
ridgecv.alpha_
```

```
3.87629874431473
```

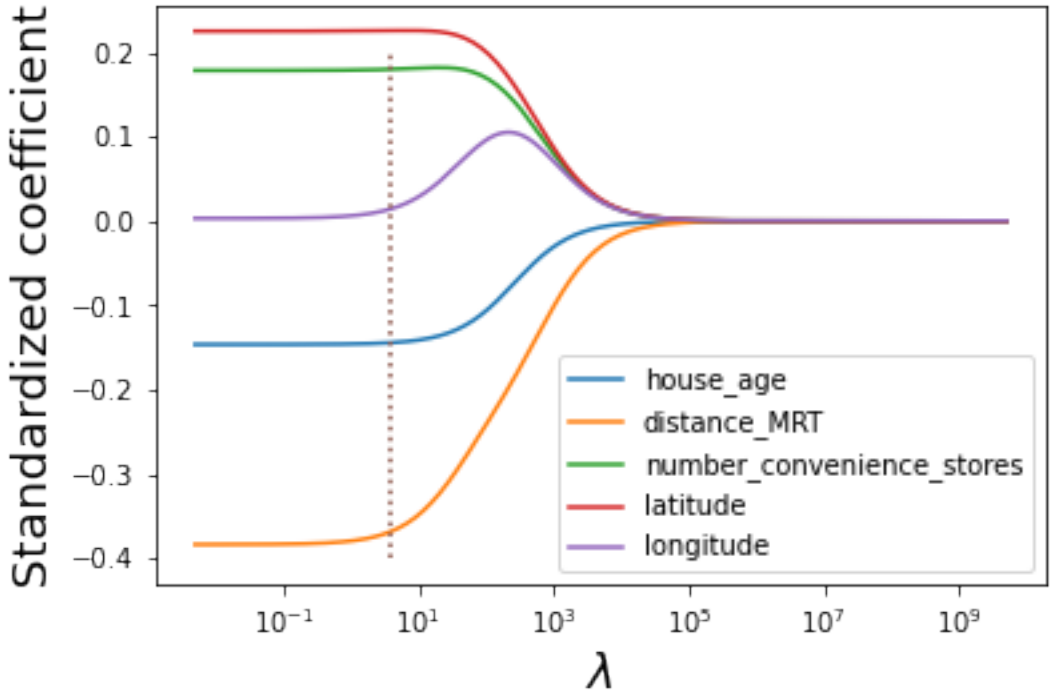
```
#Visualizing the LOOCV (leave one out cross validatation error vs lambda)
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.plot(ridgecv.alphas,ridgecv.cv_values_.sum(axis=0))
plt.plot([ridgecv.alpha_,ridgecv.alpha_],[40,40.2],':')
plt.xlabel('$\lambda$')
plt.ylabel('Cross-validation error')
```

```
Text(0, 0.5, 'Cross-validation error')
```



Note that the cross validation error is minimum at the optimal value of the tuning parameter.

```
#Visualizing the shrinkage in ridge regression coefficients with increasing values of the
alphas = 10**np.linspace(10,-2,200)*0.5
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.plot(alphas, coefs)
plt.plot([ridgecv.alpha_,ridgecv.alpha_],[-0.4,0.2],':')
plt.xscale('log')
plt.xlabel('$\lambda$')
plt.ylabel('Standardized coefficient')
plt.legend(train.columns[1:6]);
```



10.1.3 RMSE on test data

```
#Test dataset
Xtest = test.iloc[:,1:6]

#Standardizing test data
Xtest_std = scaler.transform(Xtest)
```

```
#Using the developed ridge regression model to predict on test data
ridge = Ridge(alpha = ridgecv.alpha_)
ridge.fit(Xstd, y)
pred=ridge.predict(Xtest_std)
```

```
#RMSE on test data
np.sqrt(((np.exp(pred)-test.house_price)**2).mean())
```

```
405.6227485138042
```

Note that the RMSE is similar to the one obtained using least squares regression on all the five predictors. This is because the coefficients were required to shrink very slightly for the best ridge regression fit. This may happen when we have a low number of predictors, where most of them are significant. Ridge regression is likely to perform better than least squares in case of a large number of predictors, where an OLS model will be prone to overfitting.

10.1.4 Model coefficients & R-squared

```
#Checking the coefficients of the ridge regression model
ridge.coef_
```

```
array([-0.1444778 , -0.36856553,  0.17986479,  0.22566444,  0.01413125])
```

Note that none of the coefficients are shrunk to zero. The coefficient of `longitude` is smaller than the rest, but not zero.

```
#R-squared on train data for the ridge regression model
r2_score(ridge.predict(Xstd),y)
```

```
0.6994484432136066
```

```
#R-squared on test data for the ridge regression model
r2_score(pred,np.log(test.house_price))
```

```
0.7573027646359806
```

10.2 Lasso

Let us develop a lasso model to predict house price based on the five house features.

10.2.1 Standardizing the predictors

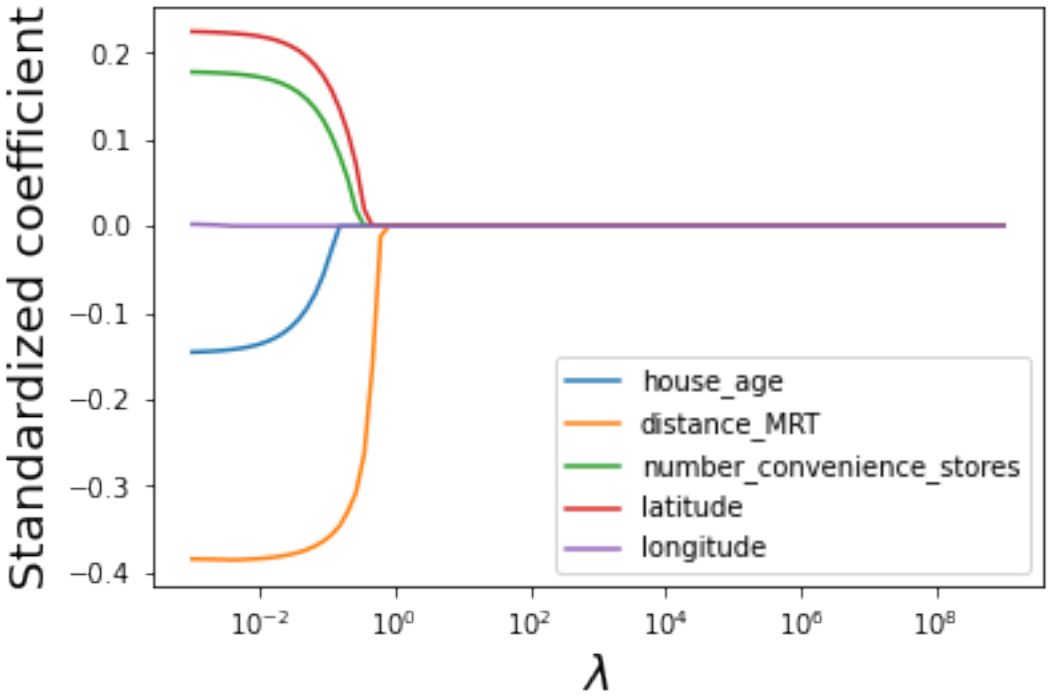
We have already standardized the predictors in the previous section. The standardized predictors are the NumPy array object `Xstd`.

10.2.2 Optimizing the tuning parameter

```
#Creating a range of values of the tuning parameter to visualize the lasso coefficients
#for different values of the tuning parameter
alphas = 10**np.linspace(10,-2,100)*0.1
```

```
#Finding the lasso coefficients for increasing values of the tuning parameter
lasso = Lasso(max_iter = 10000)
coefs = []

for a in alphas:
    lasso.set_params(alpha=a)
    lasso.fit(Xstd, y)
    coefs.append(lasso.coef_)
```

```
#Visualizing the shrinkage in lasso coefficients with increasing values of the tuning para
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.plot(alphas, coefs)
plt.xscale('log')
plt.xlabel('$\lambda$')
plt.ylabel('Standardized coefficient')
plt.legend(train.columns[1:6]);
```



Note that lasso performs variable selection. For certain values of lambda, some of the predictor coefficients are zero, while others are non-zero. This is different than ridge regression, which only shrinks the coefficients, but doesn't do variable selection.

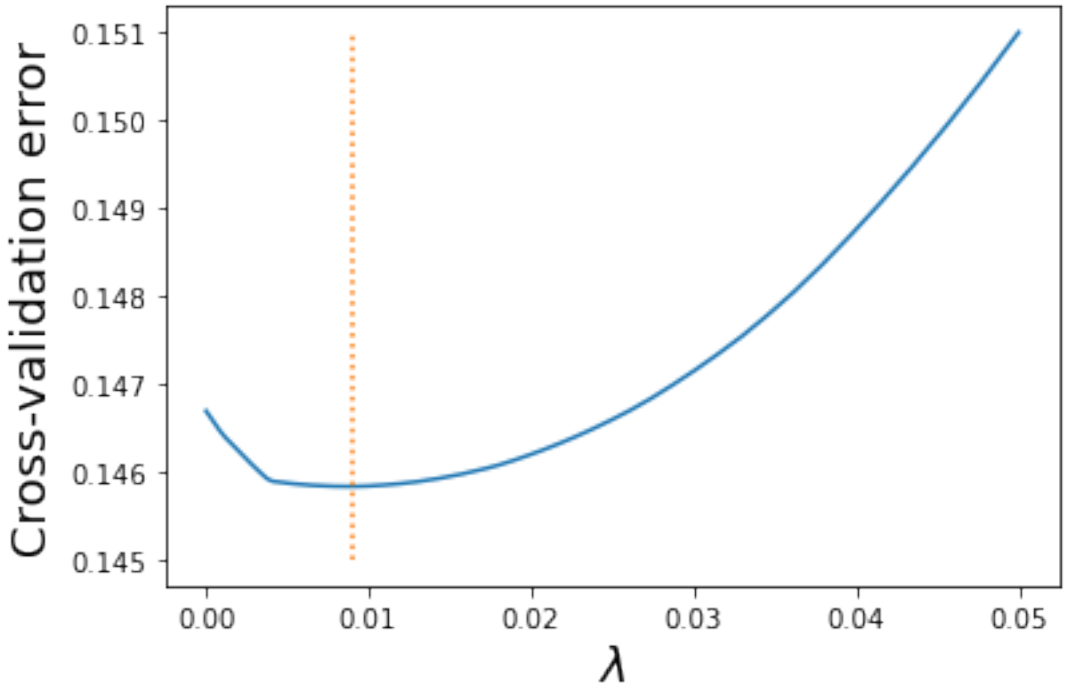
```
#Let us use cross validation to find the optimal value of the tuning parameter - lambda
#For the optimal lambda, the cross validation error will be the least

#Note that we are reducing the range of alpha so as to better visualize the minimum
alphas = 10**np.linspace(-1,-5,200)*0.5
lassocv = LassoCV(alphas = alphas, cv = 10, max_iter = 100000)
lassocv.fit(Xstd, y)

#Optimal value of the tuning parameter - lamda
lassocv.alpha_
```

```
0.009020932046960358
```

```
#Visualizing the LOOCV (leave one out cross validatation error vs lambda)
plt.xlabel('xlabel', fontsize=18)
```

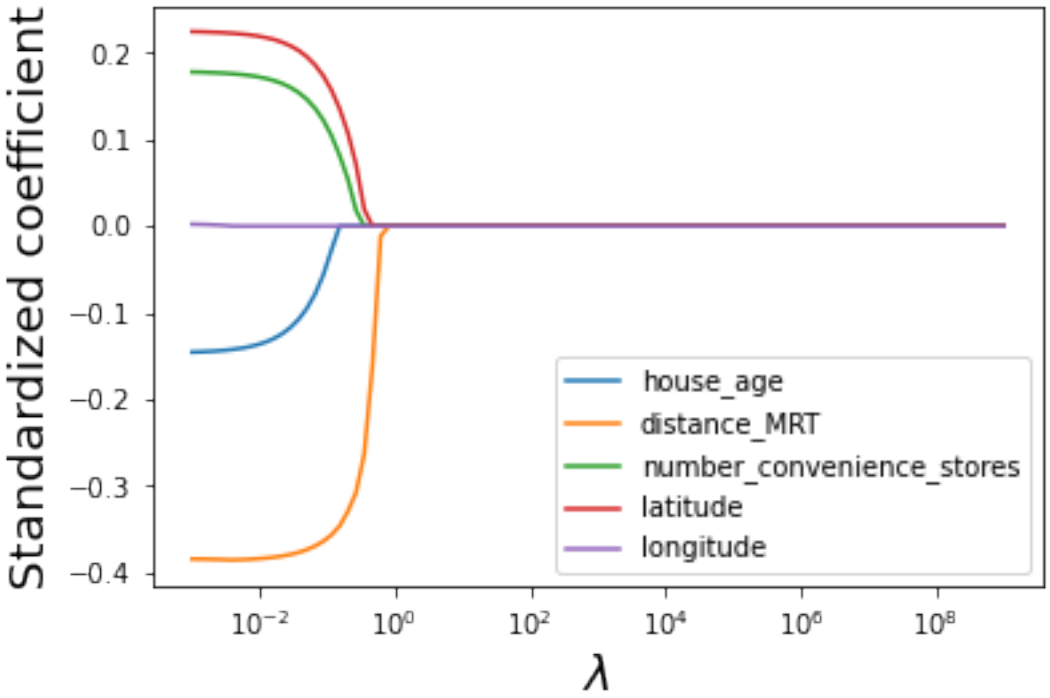
```
plt.ylabel('ylabel', fontsize=18)
plt.plot(lassocv.alphas_,lassocv.mse_path_.mean(axis=1))
plt.plot([lassocv.alpha_,lassocv.alpha_],[0.145,0.151],':')
plt.xlabel('$\lambda$')
plt.ylabel('Cross-validation error')
```

```
Text(0, 0.5, 'Cross-validation error')
```



The 10-fold cross validation error minimizes at lambda = 0.009.

```
#Visualizing the shrinkage in lasso coefficients with increasing values of the tuning para
alphas = 10**np.linspace(10,-2,100)*0.1
plt.xlabel('xlabel', fontsize=18)
plt.ylabel('ylabel', fontsize=18)
plt.plot(alphas, coefs)
plt.xscale('log')
plt.xlabel('$\lambda$')
plt.ylabel('Standardized coefficient')
plt.legend(train.columns[1:6]);
```



10.2.3 RMSE on test data

```
#Using the developed lasso model to predict on test data
lasso = Lasso(alpha = lassocv.alpha_)
lasso.fit(Xstd, y)
pred=lasso.predict(Xtest_std)
```

```
#RMSE on test data
np.sqrt(((np.exp(pred)-test.house_price)**2).mean())
```

```
400.77289943396534
```

10.2.4 Model coefficients & R-squared

```
#Checking the coefficients of the lasso model
lasso.coef_
```

```
array([-0.13720237, -0.38405197,  0.17252859,  0.21949239,  0.        ])
```

Note that the coefficient of `longitude` is shrunk to zero. Lasso performs variable selection.

```
#R-squared on train data for the lasso model
r2_score(lasso.predict(Xstd),y)
```

```
0.692606850601813
```

```
#R-squared on test data for the lasso model
r2_score(pred,np.log(test.house_price))
```

```
0.7524177148260849
```

A Datasets, assignment and project files

Datasets used in the book, assignment files, project files, and prediction problems report tempate can be found here

References