Data Science II with python (Class notes)

STAT 303-2-Sec20&21

2025-01-07

Table of contents

Preface

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

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

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

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

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

1 Simple Linear Regression

Read section 3.1 of the book before using these notes.

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

1.1 Simple Linear Regression

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

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

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

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

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

1.1.1 Training with `statsmodels`

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

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

```
# Let's create the model

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

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

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

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

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

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

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

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

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

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

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

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

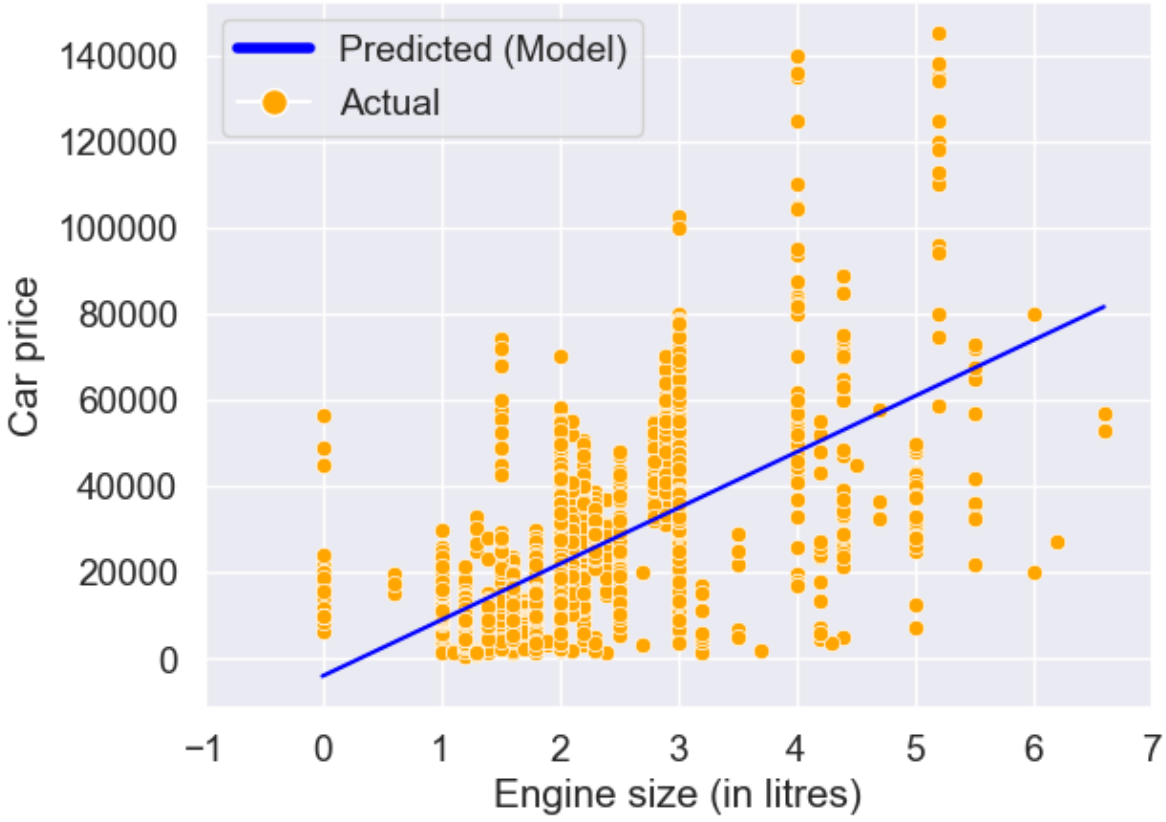
The coefficients can also be returned directly usign the `params` attribute of the `model` object returned by the `fit()` method of the `ols` class:

```
model.params
```

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

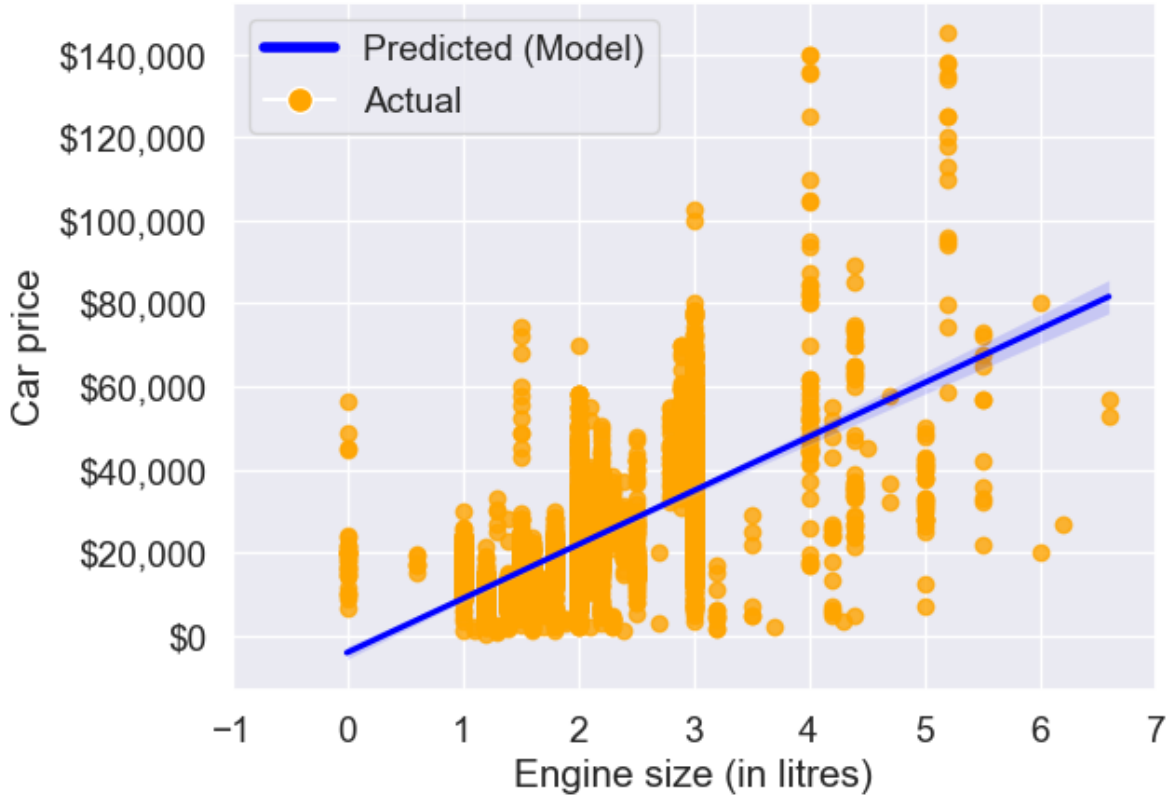
Visualize the regression line

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



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

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



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

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

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

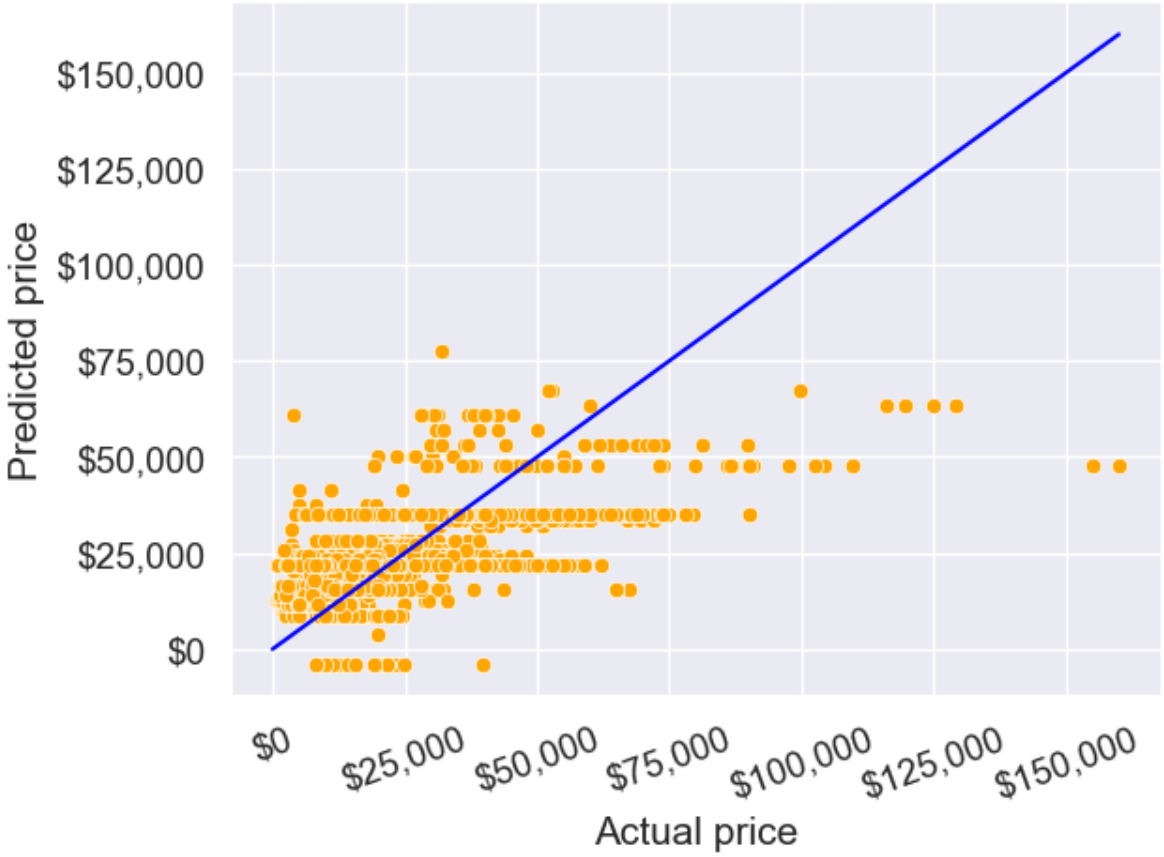
```
# Read the test data
testf = pd.read_csv('./Datasets/Car_features_test.csv') # Predictors
```

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

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

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

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



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

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

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

```
12995.106451548696
```

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

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

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

```
12810.109175214138
```

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

Find the confidence and prediction intervals of the predicted car price

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

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

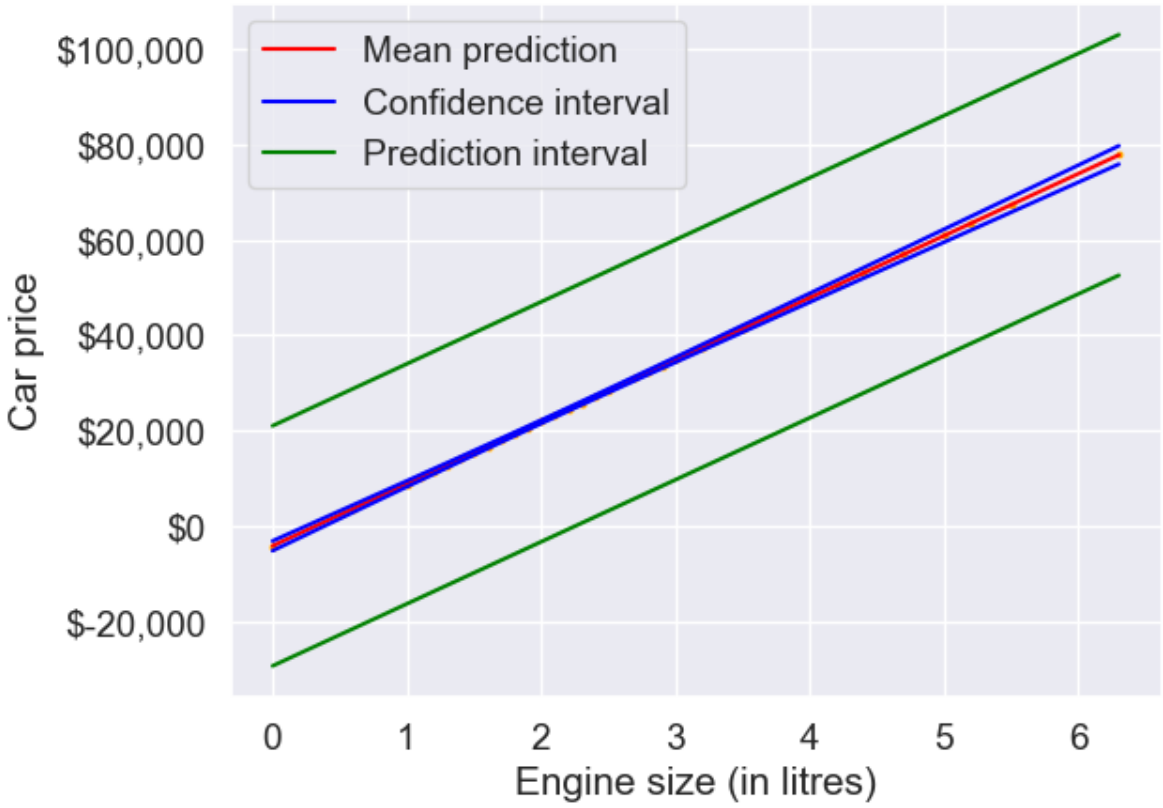
	mean	mean_se	mean_ci_lower	mean_ci_upper	obs_ci_lower	obs_ci_upper
0	34842.807319	271.666459	34310.220826	35375.393812	9723.677232	59961.937406
1	34842.807319	271.666459	34310.220826	35375.393812	9723.677232	59961.937406
2	34842.807319	271.666459	34310.220826	35375.393812	9723.677232	59961.937406
3	8866.245277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

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

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

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

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

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



1.1.2 Training with `sklearn`

```
# Create the model as an object

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

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

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

model.fit(X_train, y_train)

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

```
    # 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
```

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

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

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

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

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

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

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

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

1.1.3 Training with `statsmodels.api`

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

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

```
# Create the model as an object

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

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

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

```
# 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

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

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

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

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

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

2.1.1 Training the model

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

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

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

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

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

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

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

```
model = LinearRegression()

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

model.fit(X_train,y_train)
```

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

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

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

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

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

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

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

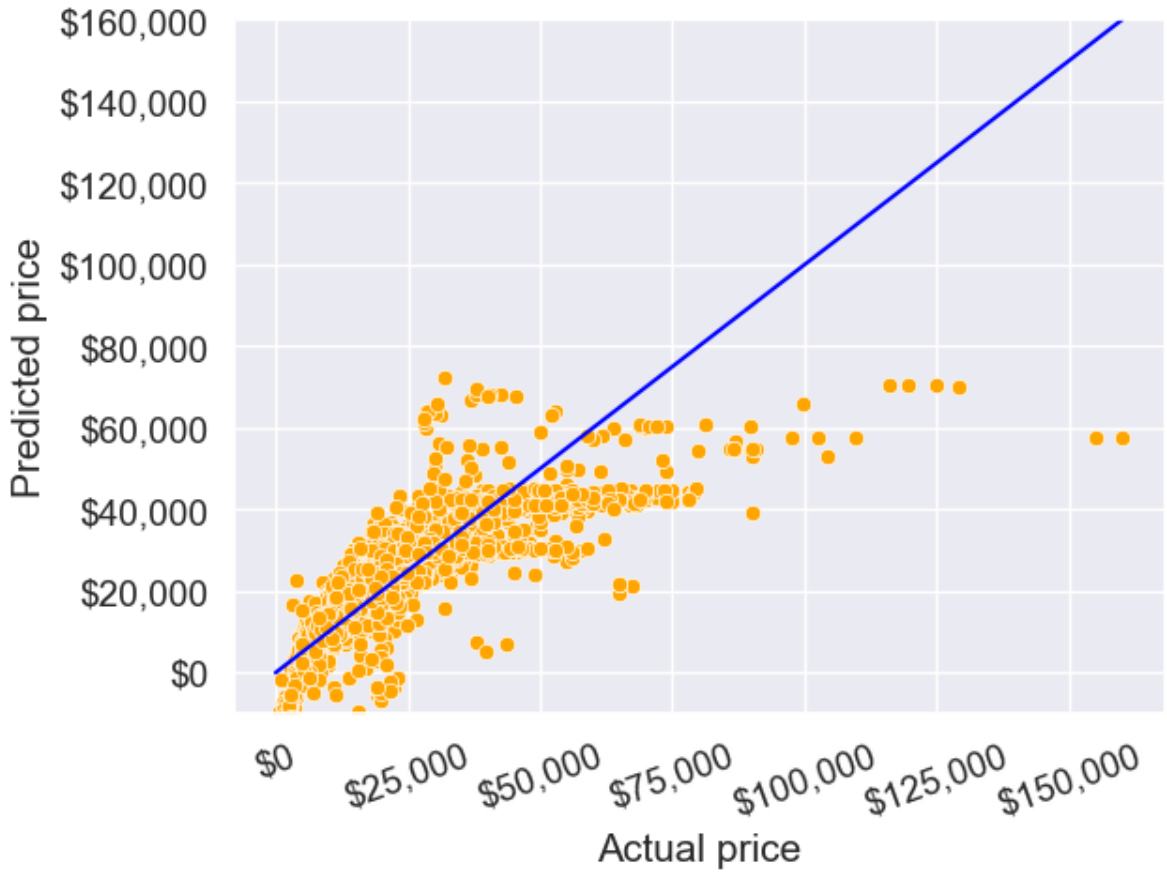
```
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
```

2.1.3 Prediction

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

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

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



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

What is the RMSE of the predicted car price?

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

```
9956.82497993548
```

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

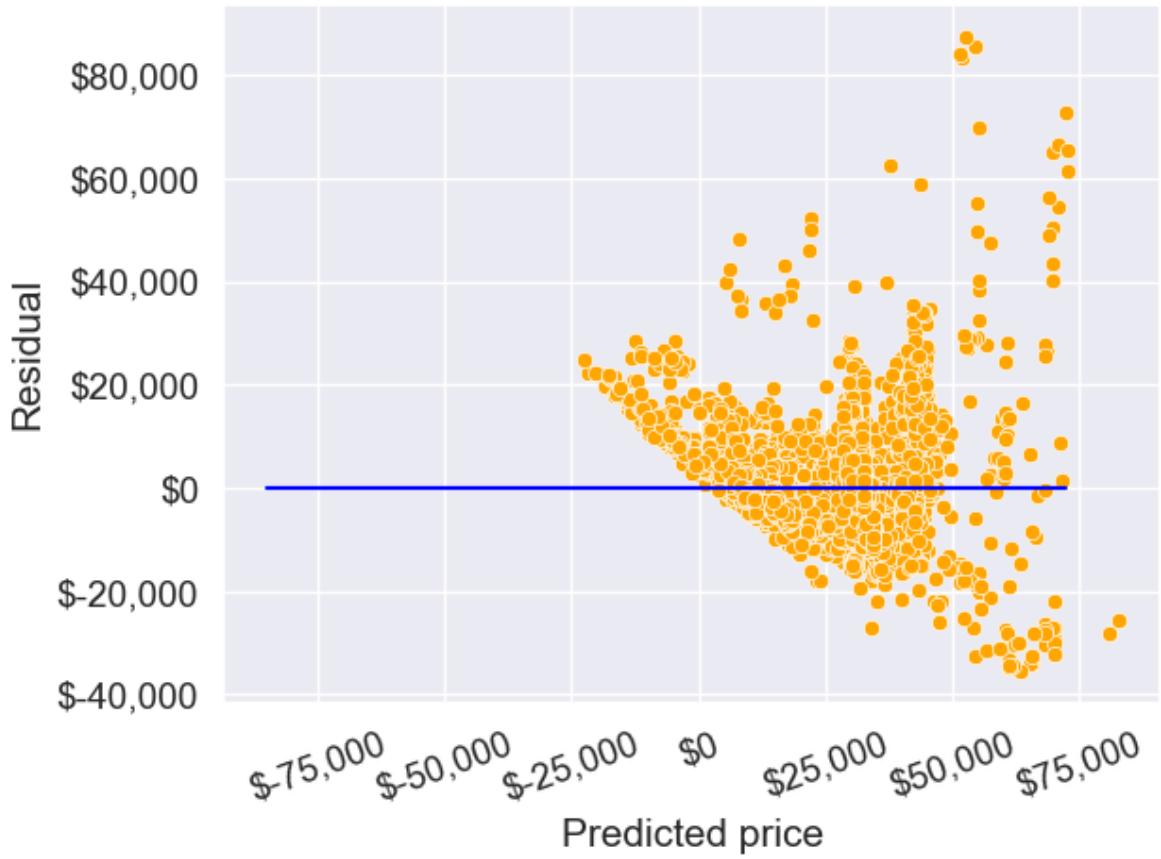
```
np.sqrt(model.mse_resid)
```

```
9563.74782917604
```

```
trainp.describe()
```

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

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



2.1.4 Effect of adding noisy predictors on R^2

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

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

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

Table 2.3: OLS Regression Results

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

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

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

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

3 Extending Linear Regression (statsmodels)

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

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

3.1 Variable interactions

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

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

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

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

3.1.1 Variable interaction between continuous predictors

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

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

Table 3.2: OLS Regression Results

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

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

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

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

The model equation is:

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

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`

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

Similarly, the equation can be re-arranged as:

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

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

However, for cars with an engine size of 3 litres, the average increase in price for a one year newer model is -275.3833+896.7687 * 3 ≈ \$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

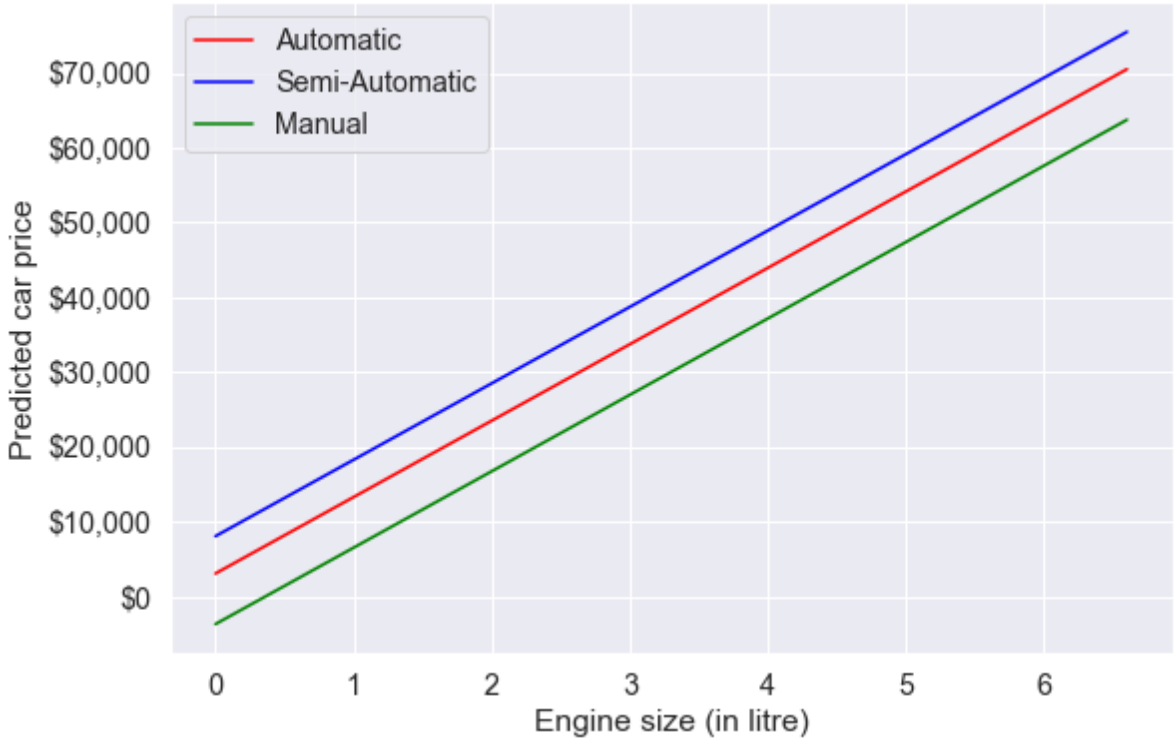
Ans: For the hypothetical scenario of a car with zero engine size and *Automatic* transmission, the estimated mean car price is ≈ \$3042.

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

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

Let us visualize the developed model.

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



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

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

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

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

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

Table 3.8: OLS Regression Results

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

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

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

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

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

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

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

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

Table 3.11: OLS Regression Results

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

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

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

The model equation for the model with interactions is:

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

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

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

or

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

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

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

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

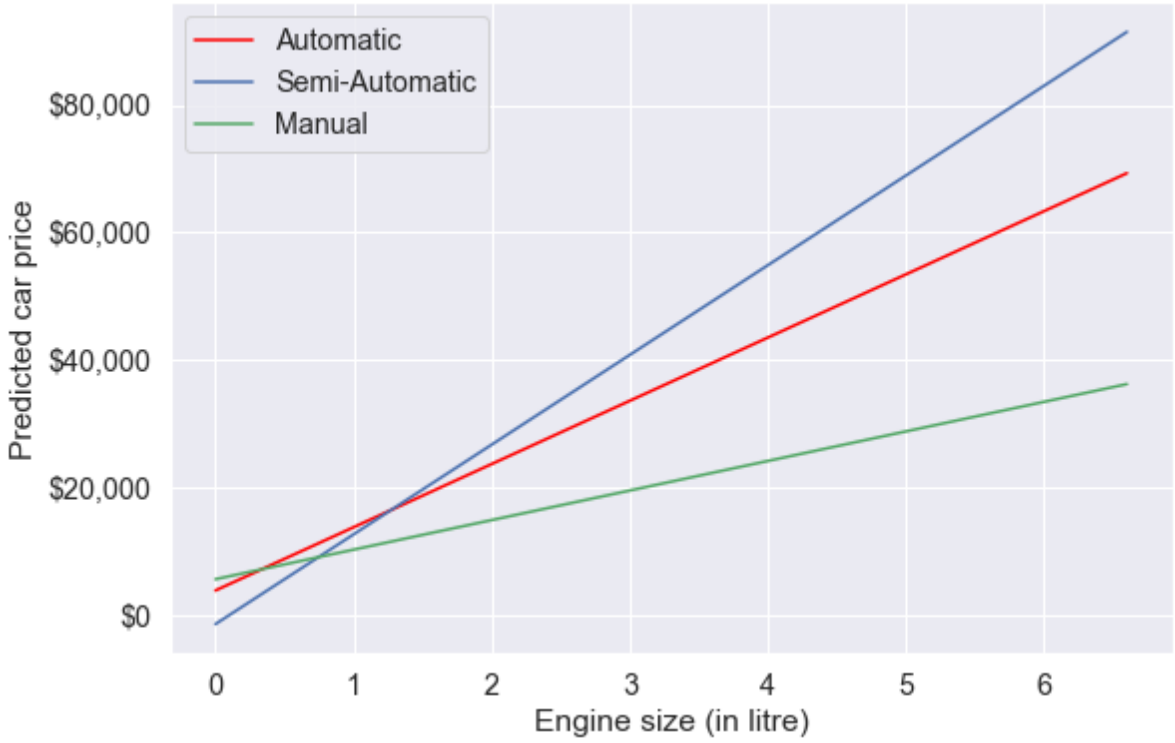
A: For a hypothetical scenario of zero engine size, the estimated mean `price` of a car with *Manual* `transmission` is ≈ \$1768 more than the estimated mean `price` of a car with *Automatic* `transmission`.

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

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

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

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



Note the interaction term adds flexibility to the model.

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

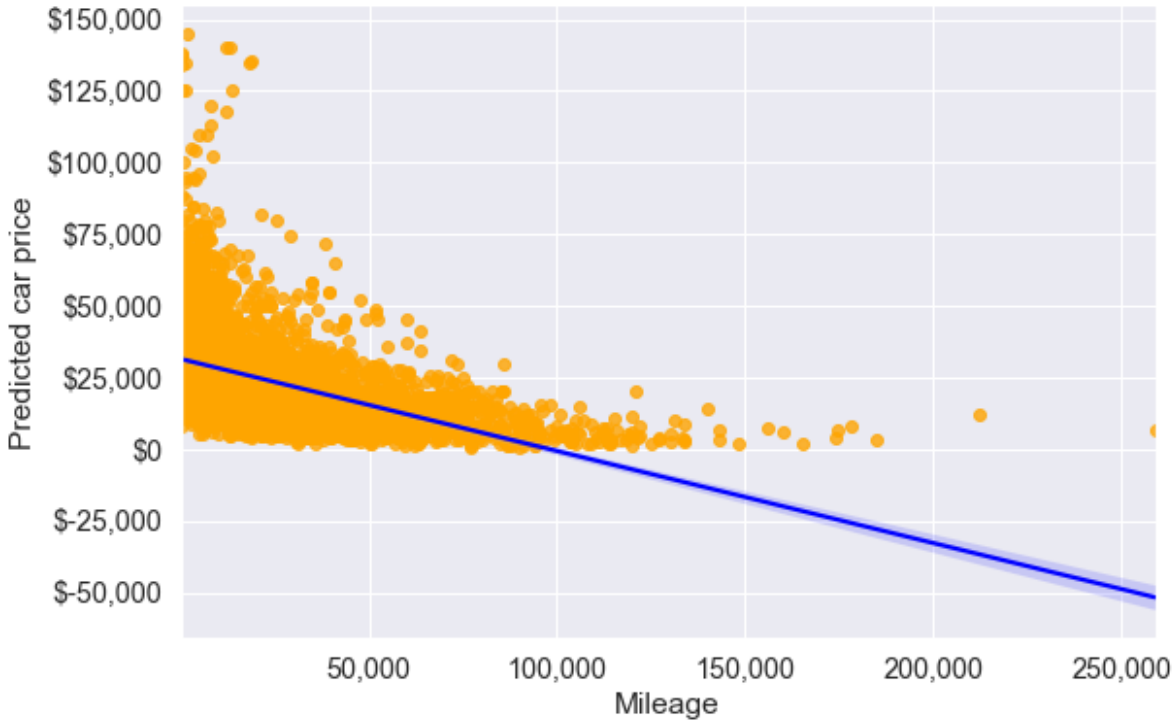
3.2 Variable transformations

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

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

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

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




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

```
0.22928048993376182
```

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

3.2.1 Quadratic transformation

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

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

Table 3.14: OLS Regression Results

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

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

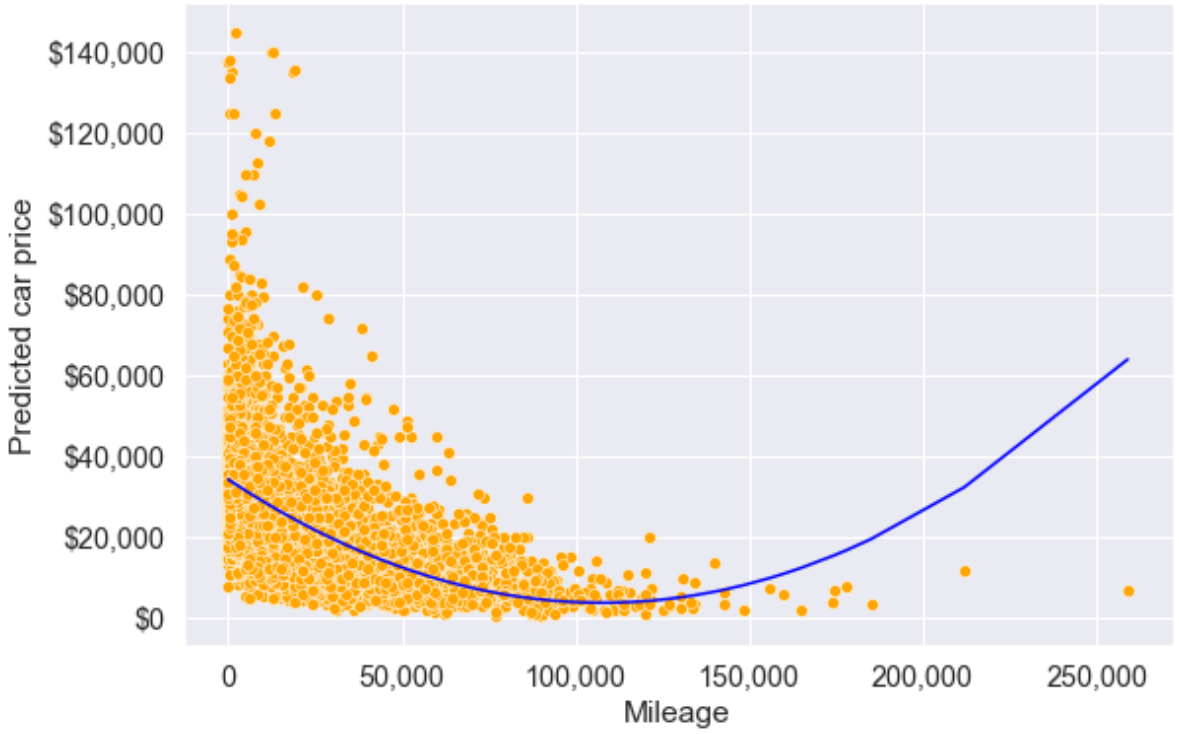
Omnibus:	2362.973	Durbin-Watson:	0.325
Prob(Omnibus):	0.000	Jarque-Bera (JB):	22427.952
Skew:	2.052	Prob(JB):	0.00
Kurtosis:	12.576	Cond. No.	4.81e+09

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

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

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

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



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

3.2.2 Cubic transformation

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

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

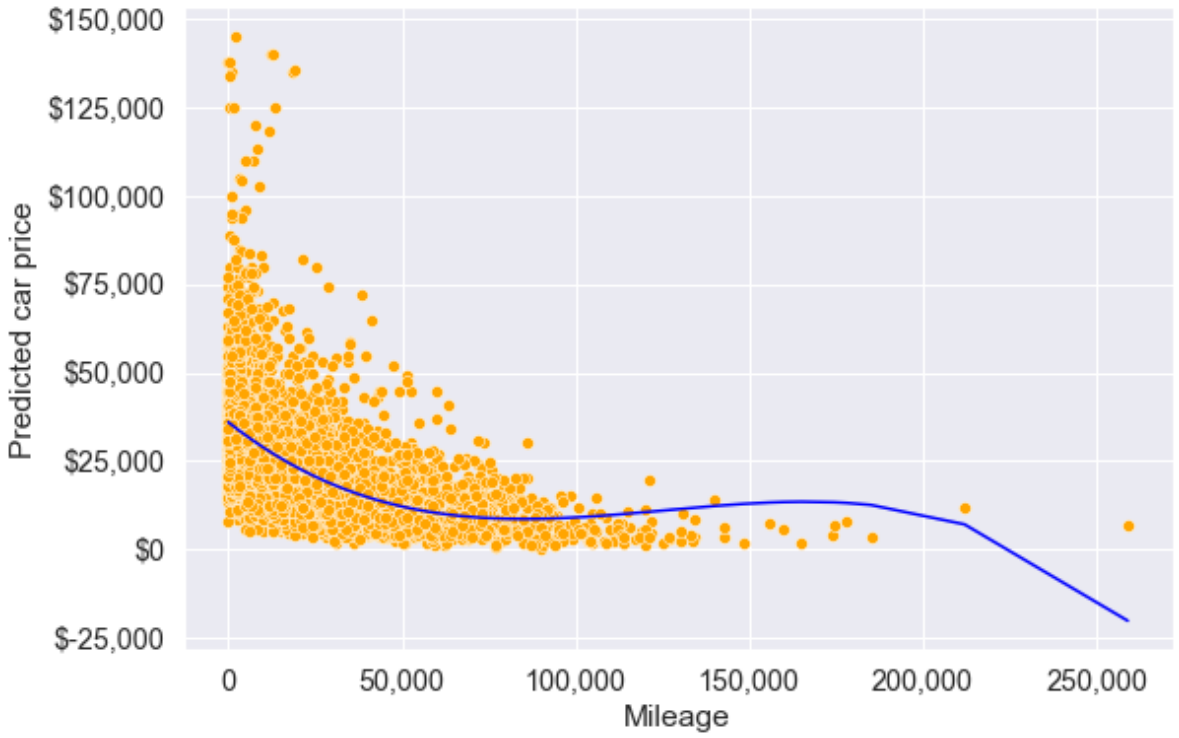
Table 3.17: OLS Regression Results

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

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

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

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



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

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

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

Table 3.20: OLS Regression Results

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

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

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

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

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

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

```
9074.494088619422
```

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

3.3 `PolynomialFeatures()`

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

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

3.3.1 Generating polynomial features

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

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

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

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

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

```
poly.get_feature_names_out()
```

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

3.3.2 Fitting the model

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

```
LinearRegression()
```

3.3.3 Testing the model

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

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

```
8896.175508213777
```

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

4 Extending Linear Regression (PolynomialFeatures in Sklearn)

4.0.1 Simulate Data

```
import numpy as np
import pandas as pd

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

# Number of samples
N = 5000

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

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

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

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

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

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

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

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

4.0.2 Train-Test Split

```
from sklearn.model_selection import train_test_split

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

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

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

4.0.3 Baseline Model (original Features)

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

# Create a linear regression model
baseline_model = LinearRegression()

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

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

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

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

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

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

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

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

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

```
from sklearn.preprocessing import PolynomialFeatures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.0.8 Putting all together

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

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

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

model_comparison
```

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

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

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

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

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

4.0.9 degreee = 4

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

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

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

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

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

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

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

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

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

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

```
R2: 0.98706356387817

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

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

```
Number of Features: 14

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

4.1 Key takeaway:

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

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

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

Use `interaction_only` for Cross Terms Only

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

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

X_transformed = poly_int.fit_transform(X)
```

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

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

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

```
import numpy as np

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

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

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

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

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

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

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

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

5 Beyond Fit (implementation)

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

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

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

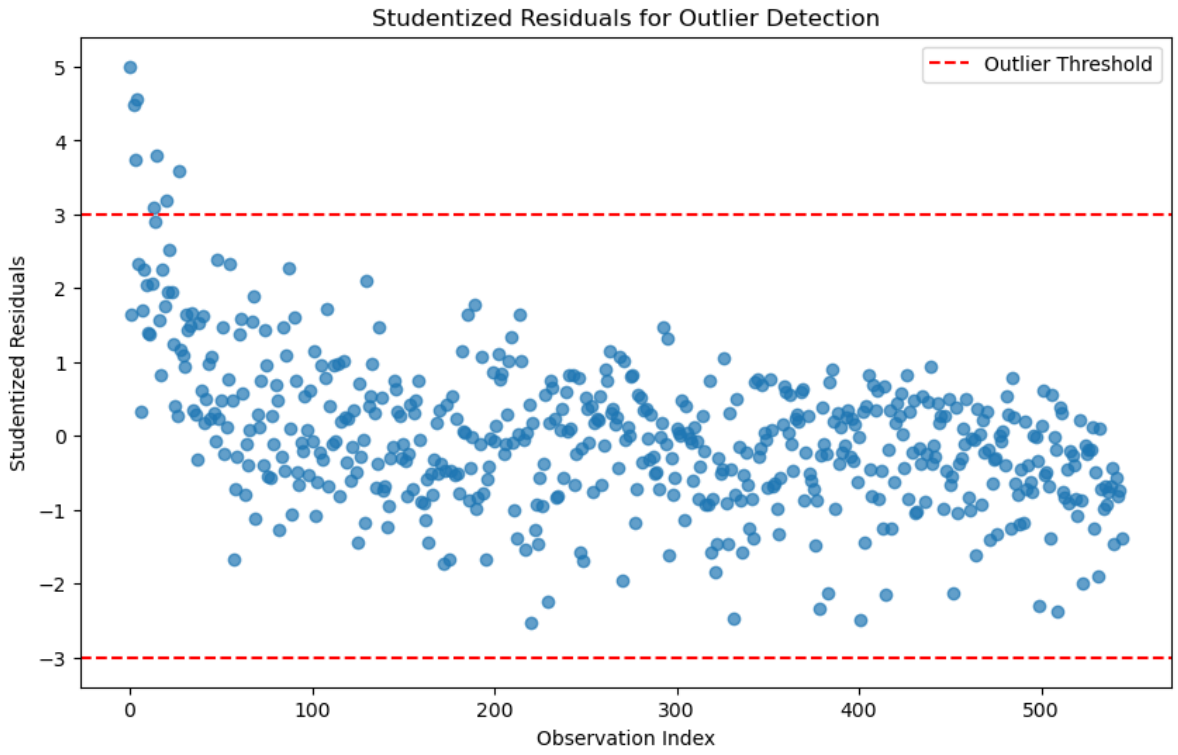
```
# build a formular api model using price as the target, the rest of the variables as predicto
model = smf.ols('price ~ area + bedrooms + bathrooms + stories + mainroad + guestroom + basem
model = model.fit()
print(model.summary())
```

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

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

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

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

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




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

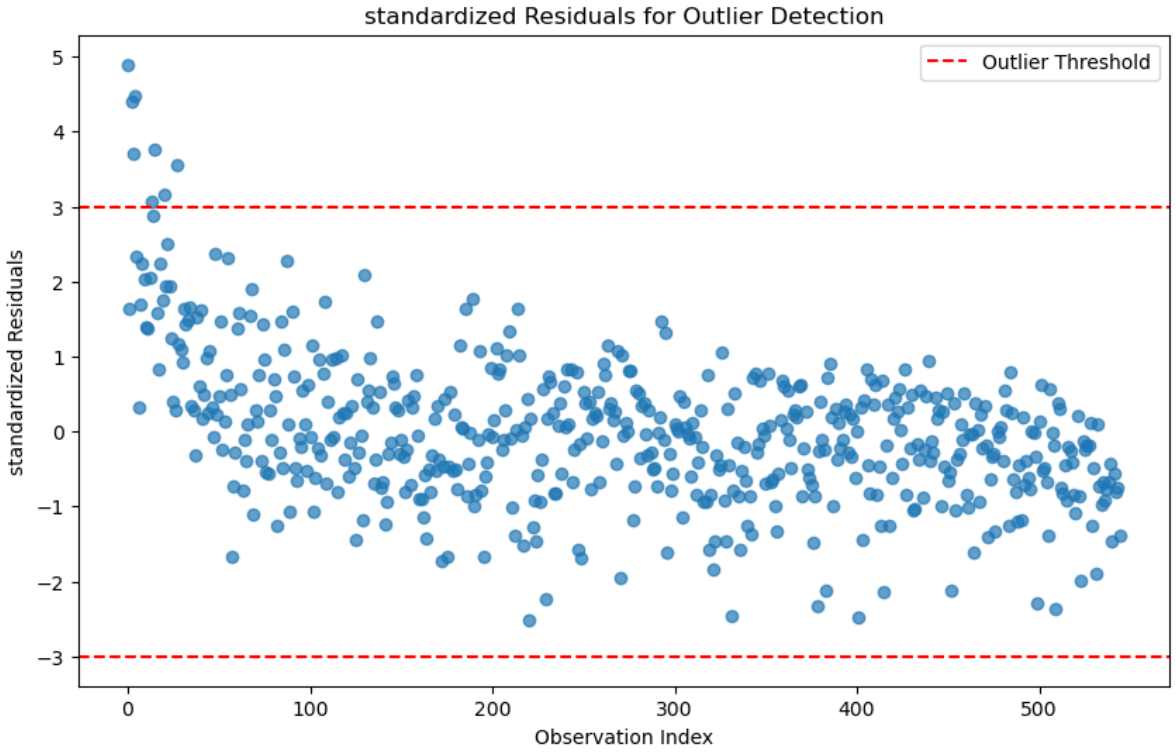
```
Outliers detected at indices: [ 0  2  3  4 13 15 20 27]
```

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

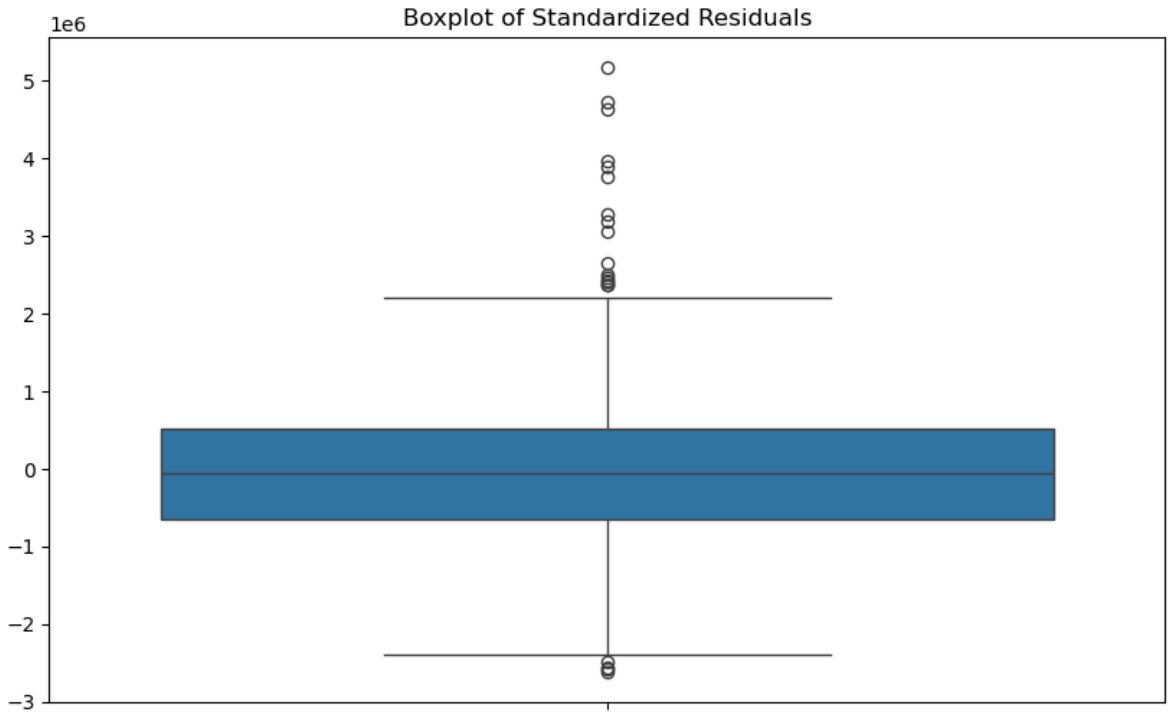
```
# Identify observations with high standardized residuals
outlier_indices_standardized = np.where(np.abs(outliers_standardized) > outlier_threshold)[0]
print(f"Outliers detected at indices: {outlier_indices_standardized}")
```

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

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




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




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

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

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

5.0.0.1 Identifying High Leverage Points

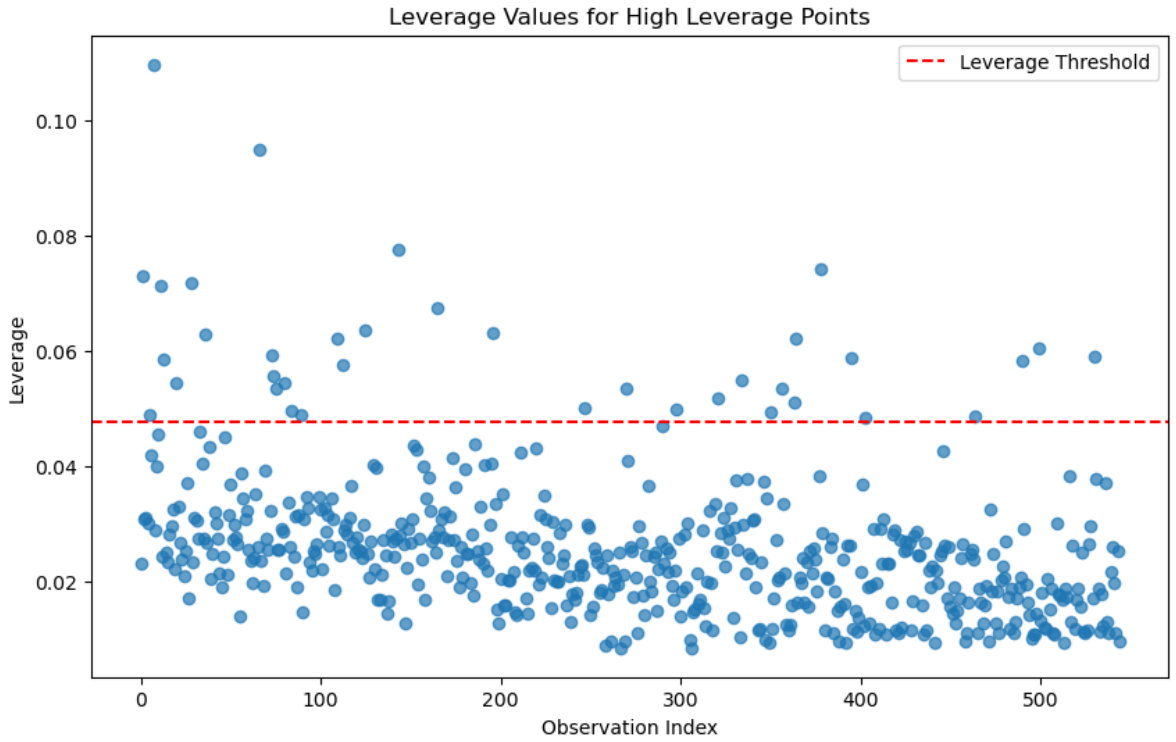
A common threshold for identifying **high leverage points** in regression analysis is:

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

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

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

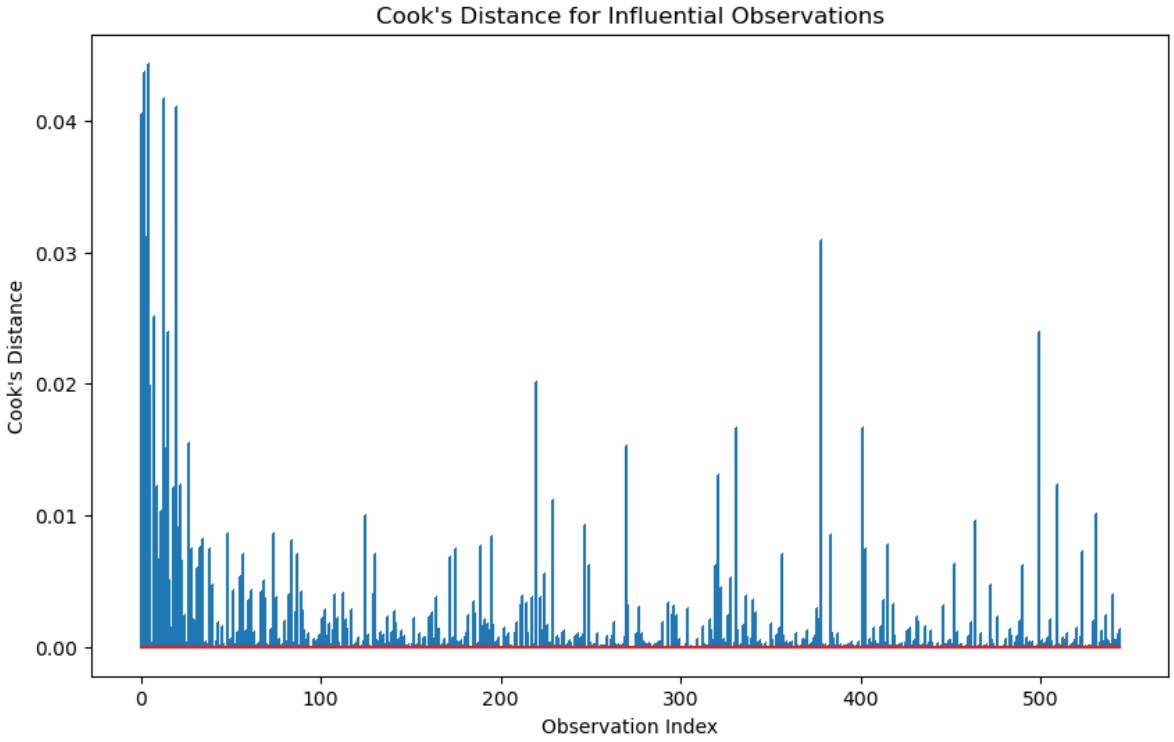
```
# Plot leverage values
plt.figure(figsize=(10, 6))
plt.scatter(range(len(leverage)), leverage, alpha=0.7)
plt.axhline(y=leverage_threshold, color='r', linestyle='--', label='Leverage Threshold')
plt.title('Leverage Values for High Leverage Points')
plt.xlabel('Observation Index')
plt.ylabel('Leverage')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




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

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

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

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



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

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

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

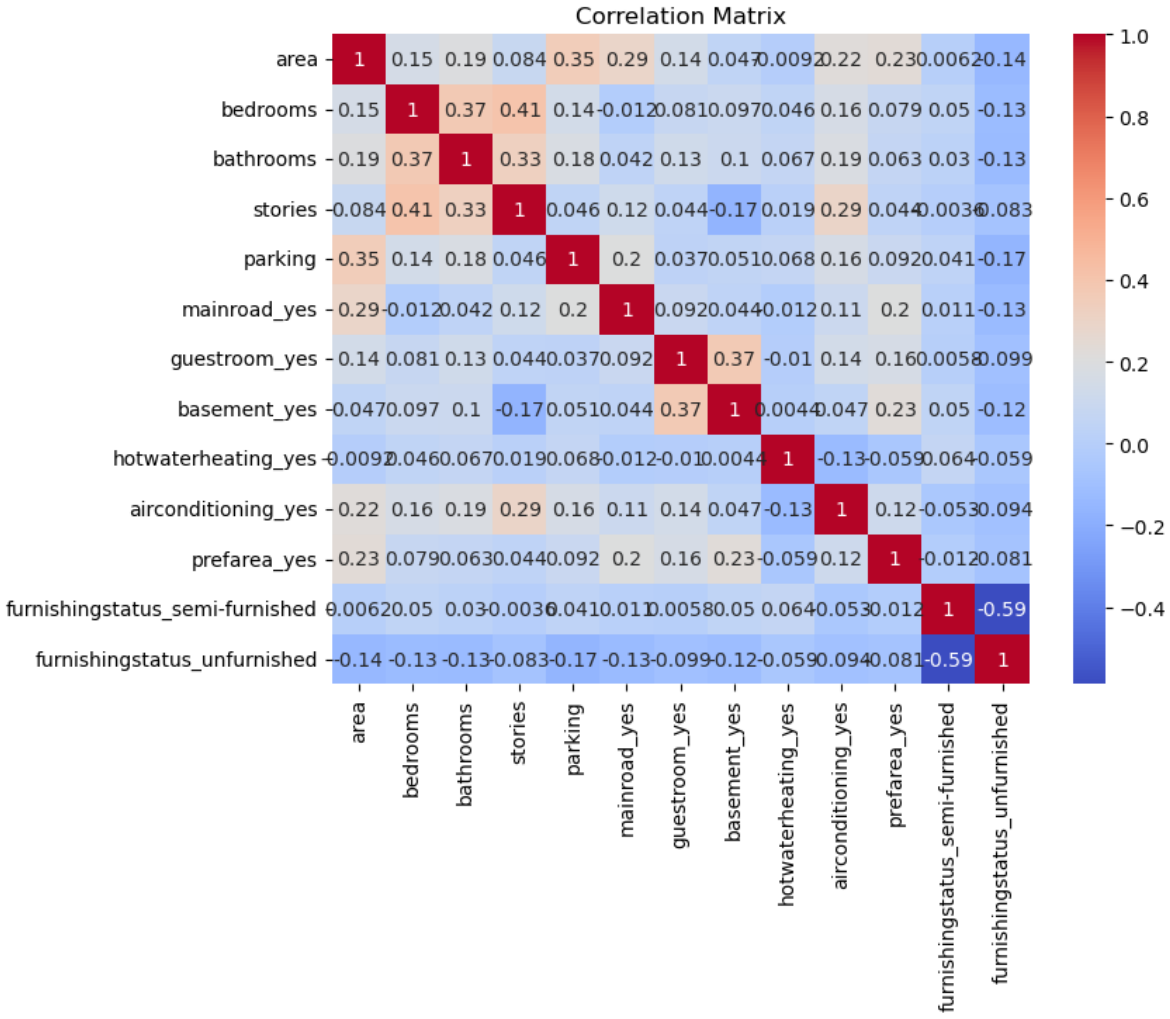
```python
# ======================================
# 4. Checking Multicollinearity (VIF)
# ======================================
# VIF calculation
from statsmodels.stats.outliers_influence import variance_inflation_factor

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

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

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


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

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

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

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

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

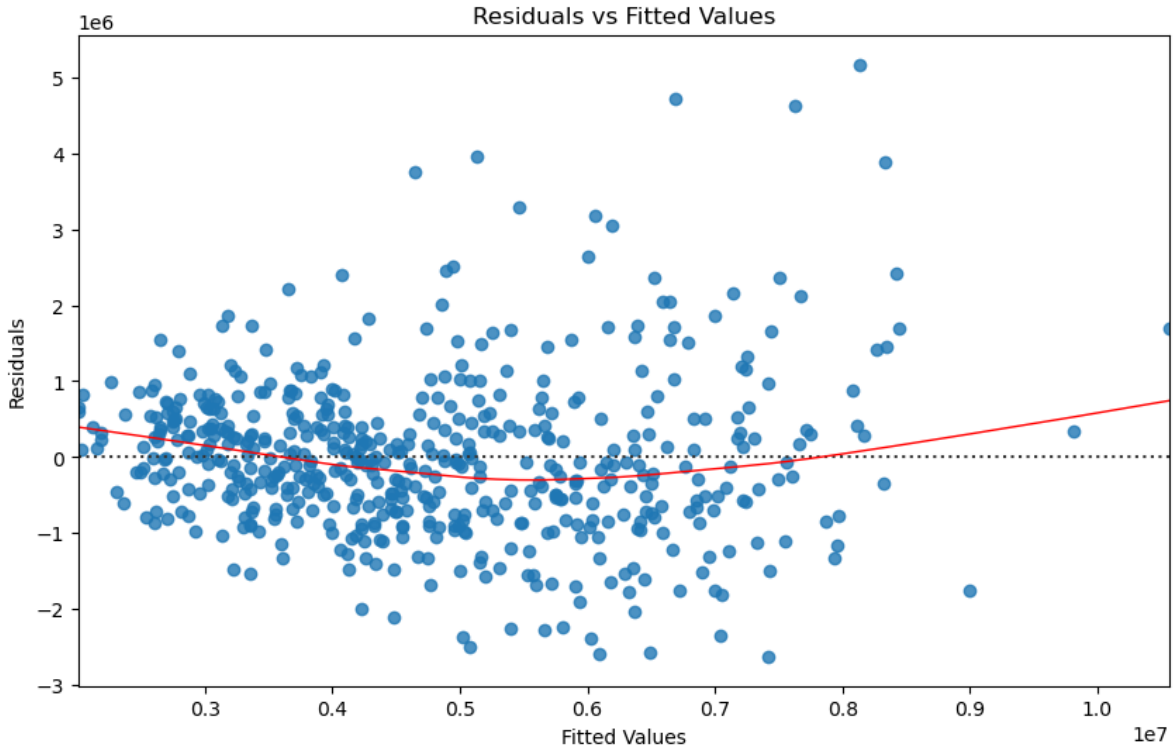



```
#output the correlation of other predictors with the bedrooms
X.corr()['bedrooms'].abs().sort_values(ascending=False)
```

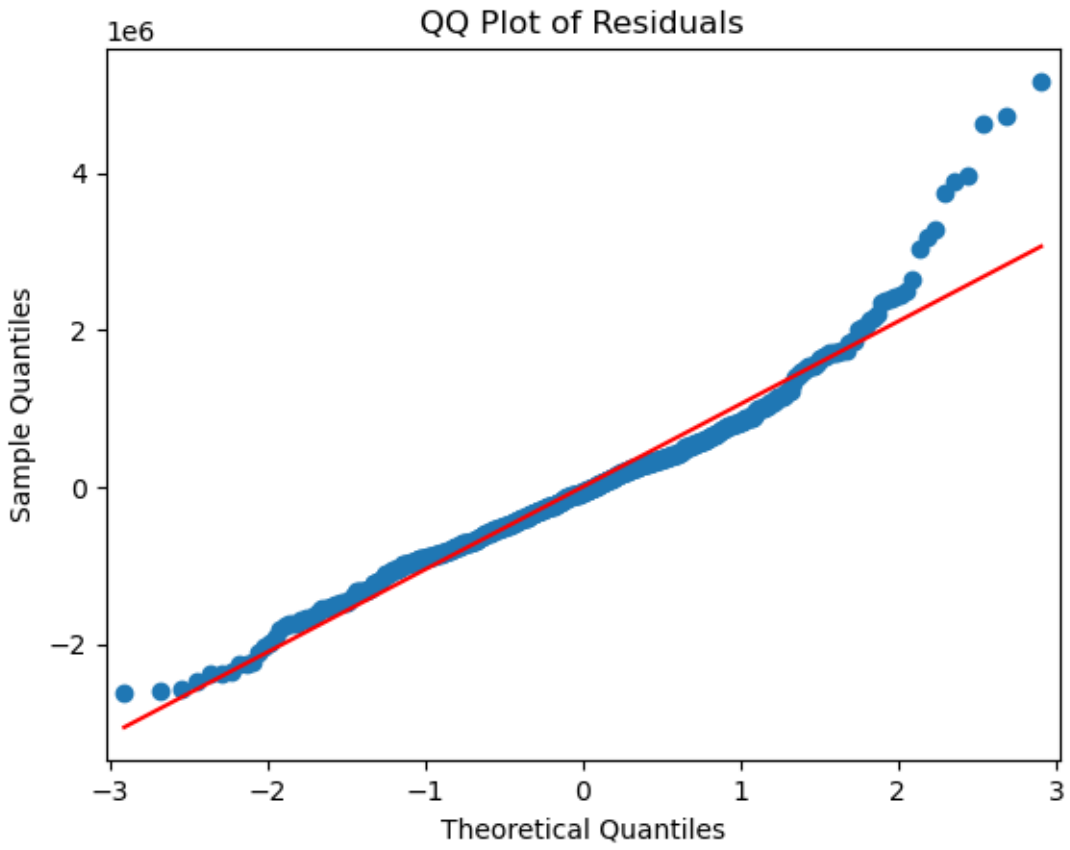
```
bedrooms                          1.000000
stories                           0.408564
bathrooms                         0.373930
airconditioning_yes               0.160603
area                              0.151858
parking                           0.139270
furnishingstatus_unfurnished      0.126252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

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

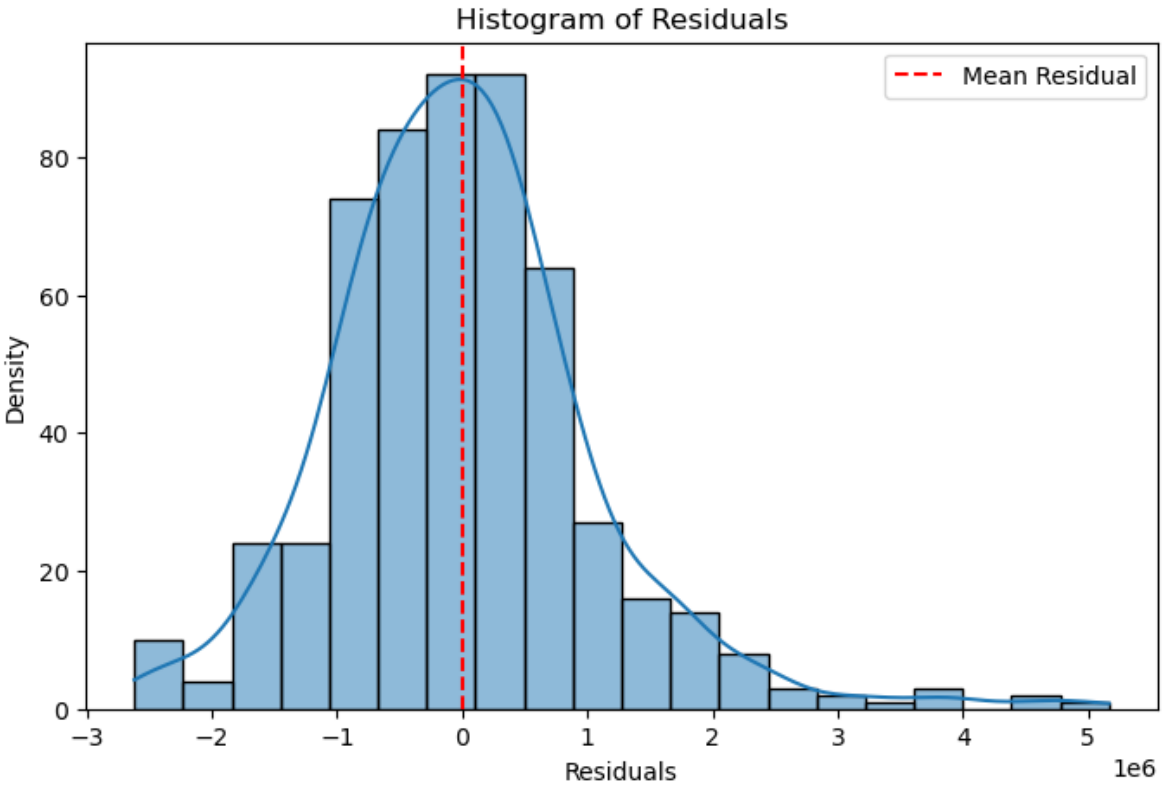
```
# ----------------------------------------
# 5. Analyzing Residual Patterns
# ----------------------------------------
# Residuals vs Fitted Values Plot
fitted_values = model.fittedvalues
residuals = model.resid

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

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



<Figure size 1000x600 with 0 Axes>




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


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

6 Beyond Fit (statistical theory)

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

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

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

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

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

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

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

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

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

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

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

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

```
9094.209473232966
```

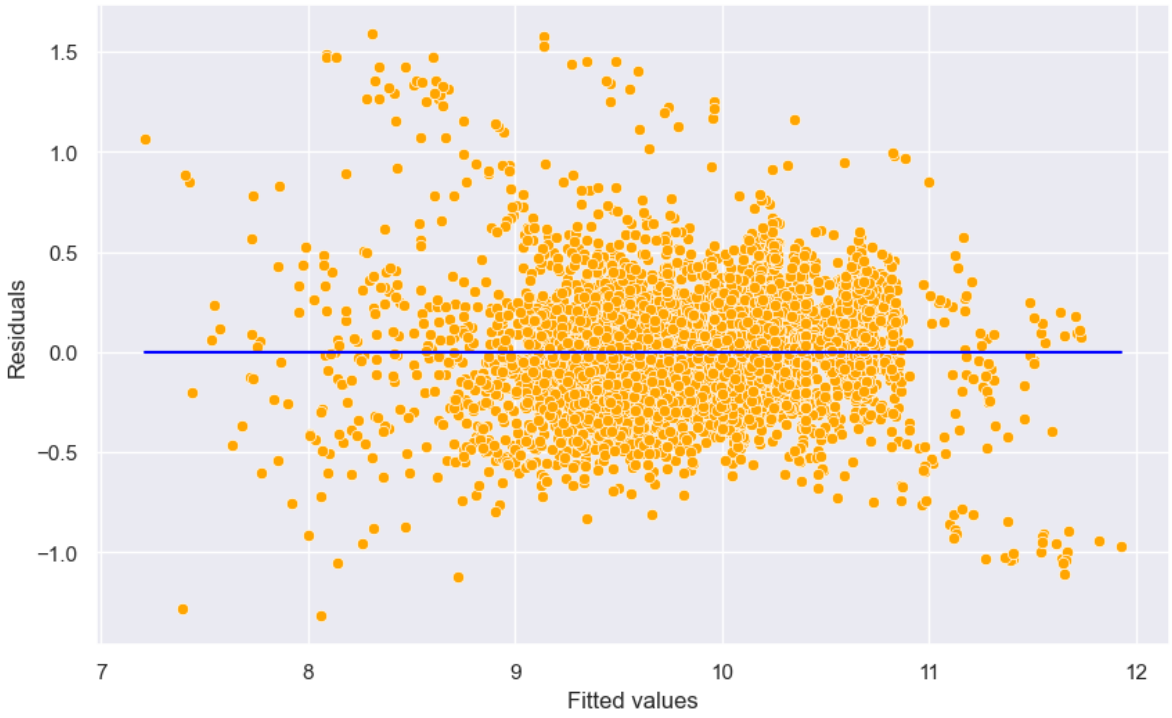
6.1 Outliers

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

```
1.9604435402730618
```

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

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

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

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

```
4.4200129981725365
```

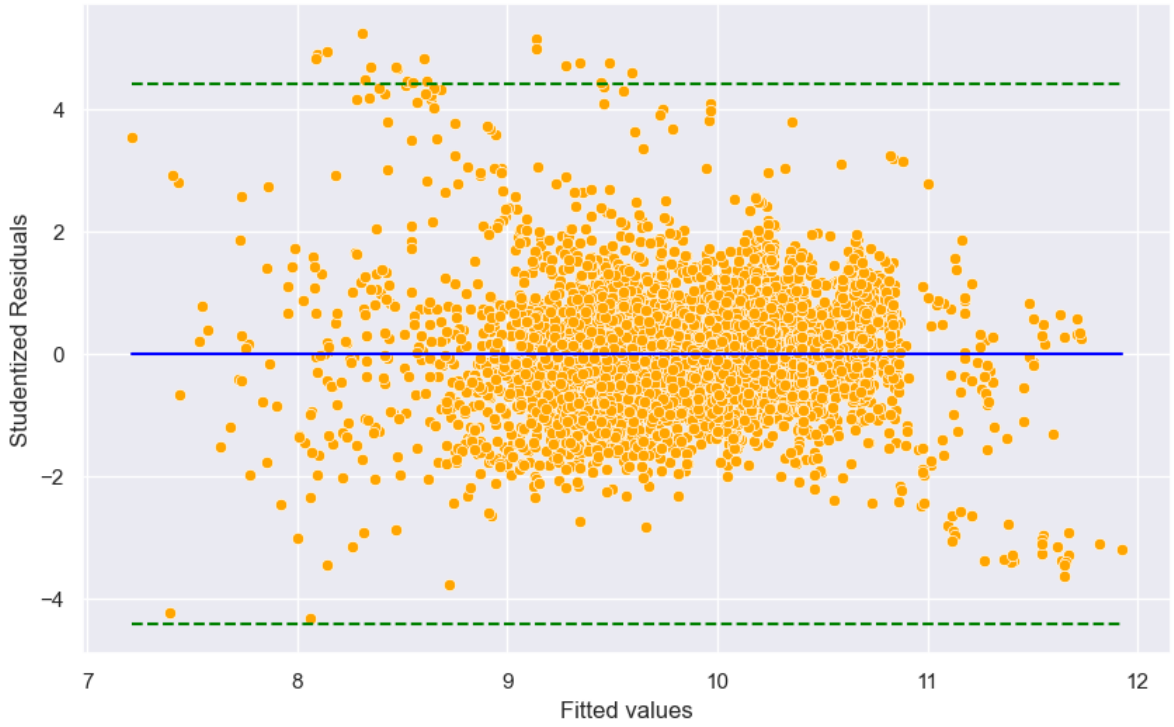
The studentized residuals can be obtained using the `outlier_test()` method of the object returned by the `fit()` method of an OLS object. Let us find the studentized residuals in our car `price` prediction model.

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

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

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

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

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



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

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

Are there outliers in our example?

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

```
19
```

Let us analyze the outliers.

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

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

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

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

6.2 High leverage points

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

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

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

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

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

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

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

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

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

Thumb rules:

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

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

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

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

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

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

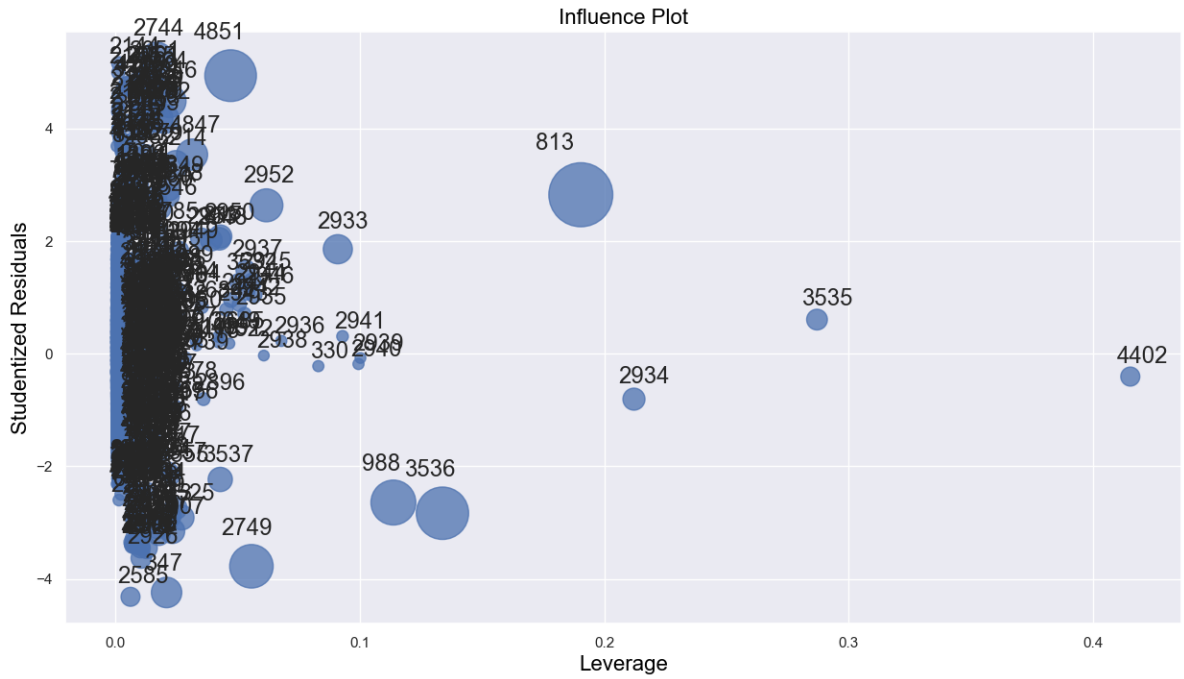
	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	-238.2125	25.790	-9.237	0.000	-288.773	-187.652
year	0.1227	0.013	9.608	0.000	0.098	0.148
engineSize	13.8349	5.795	2.387	0.017	2.475	25.195
mileage	0.0005	0.000	3.837	0.000	0.000	0.001
mpg	-1.2446	0.345	-3.610	0.000	-1.921	-0.569
year:engineSize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

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

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

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

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



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

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

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

```
0.0024193548387096775
```

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

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

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

```
269
```

6.2.1 Identifying extrapolation using leverage

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

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

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

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

```
leverage[0]
```

```
0.0026426981240353694
```

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

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

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

```
0.0026426973869101977
```

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

6.3 Influential points

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

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

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

```
train_filtered.shape
```

```
(4948, 11)
```

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

```
12
```

We removed 12 influential data points from the training data.

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

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

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

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

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

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

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

```
9811.74078331643
```

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

```
9800.202063309154
```

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

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

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

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

```
8922.977452912108
```

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

6.3.1 Influence on single fitted value (DFFITS)

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

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

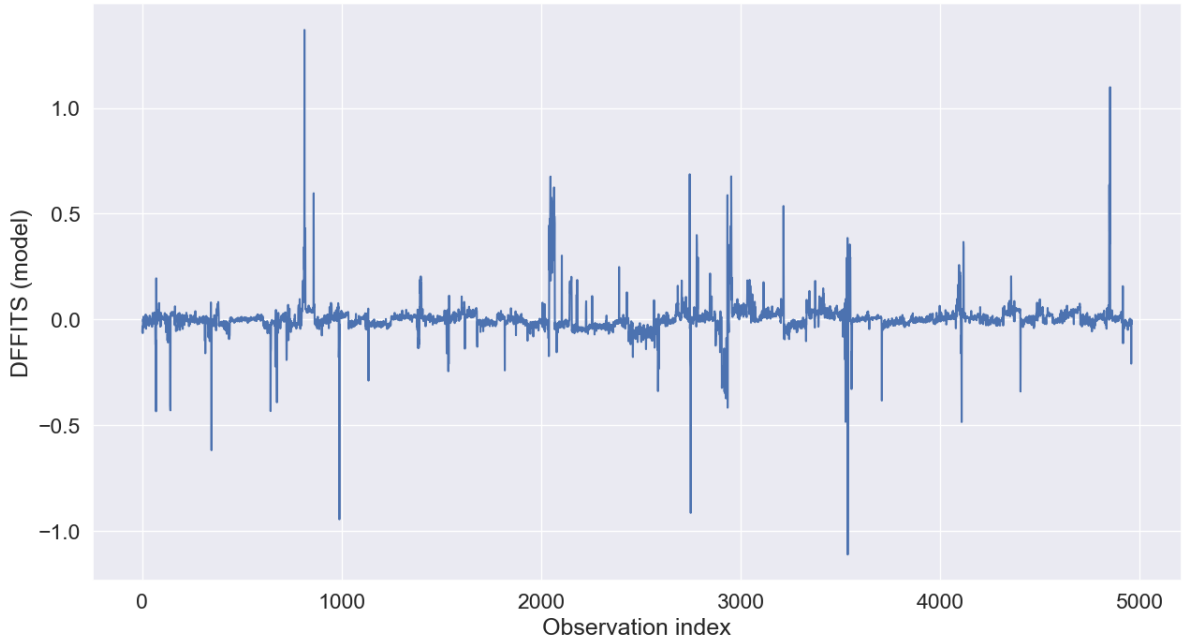
- Note that the denominator in the above fraction is the estimated standard deviation of $\hat{Y}_i$, but uses the error mean square when the i^{th} observation is omitted.
- $DFFITS$ for the i^{th} observation represents the number of estimated standard deviations of $\hat{Y}_i$ that the fitted value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

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

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

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

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



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

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

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

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

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

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

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

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

```
813    5502.647323
dtype: float64
```

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

```
813    4581.374593
dtype: float64
```

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

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

```
0.19038697461006687
```

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

```
2.823478041409651
```

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

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

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

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

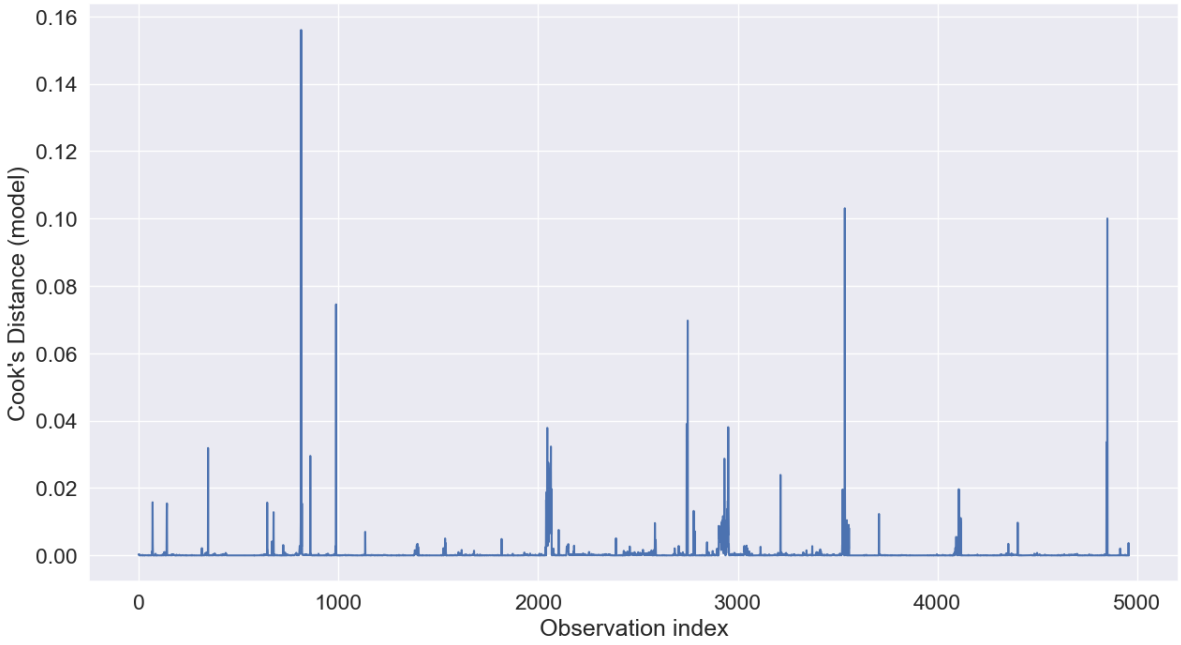
It can be shown that:

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

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

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

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




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

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

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

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

```
0.9402181103263811
```

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

6.3.3 Influence on regression coefficients (DFBETAS)

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

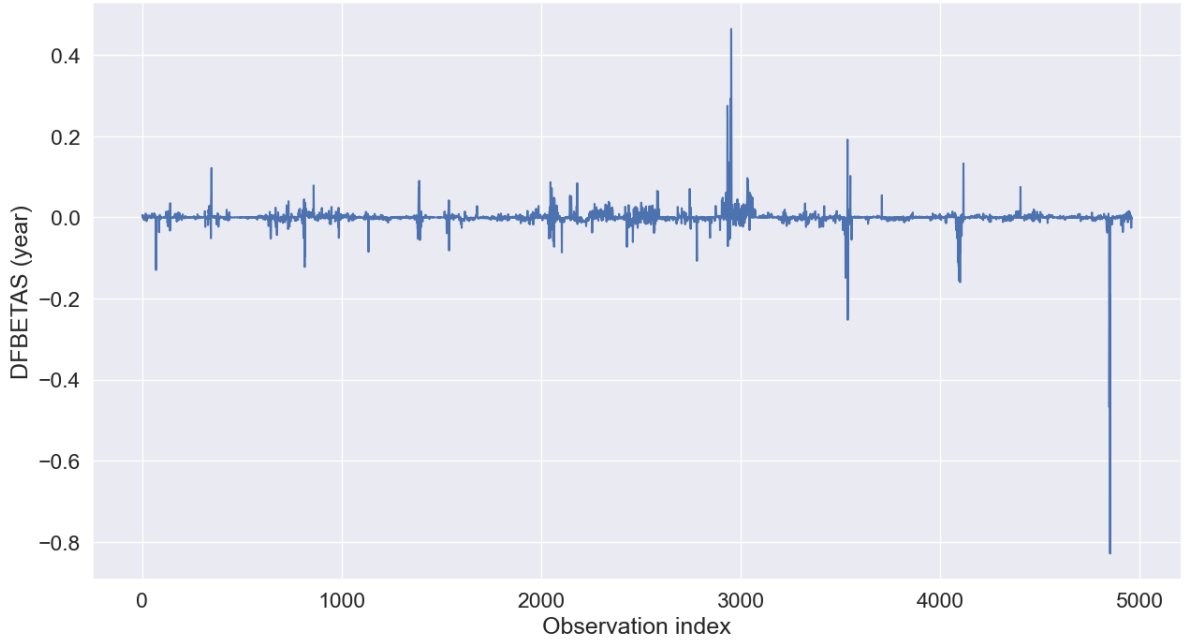
$$(DFBETAS)_{k(i)} = \frac{\hat{\beta}_k - \hat{\beta}_{k(i)}}{\sqrt{MSE_i c_k}} \tag{6.7}$$

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

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

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

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



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

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

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

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

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

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

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

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

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

```
0.047120455781282225
```

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

```
4.938606329343604
```

Do you see what makes this point influential?

6.4 Collinearity

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

6.4.1 Why and how is collinearity a problem

(Source: page 100-101 of book)

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

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

6.4.2 How to measure collinearity/multicollinearity

(Source: page 102 of book)

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

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

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

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:

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

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

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

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

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

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

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

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

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

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

print(vif_data)
```

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

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

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

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

6.4.3 Manual computation of VIF

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

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

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

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

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

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

```
2.044183378279958
```

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

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

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

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

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

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

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

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

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

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

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

```
2.4509803921568625
```

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

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

Table 6.6: OLS Regression Results

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

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

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

Removing mileage from the above regression.

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

Table 6.9: OLS Regression Results

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

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

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

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

6.4.4 When can we overlook multicollinearity?

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

7 Logistic regression: Introduction and Metrics

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

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

7.1 Theory Behind Logistic Regression

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

7.1.1 Description

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

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

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

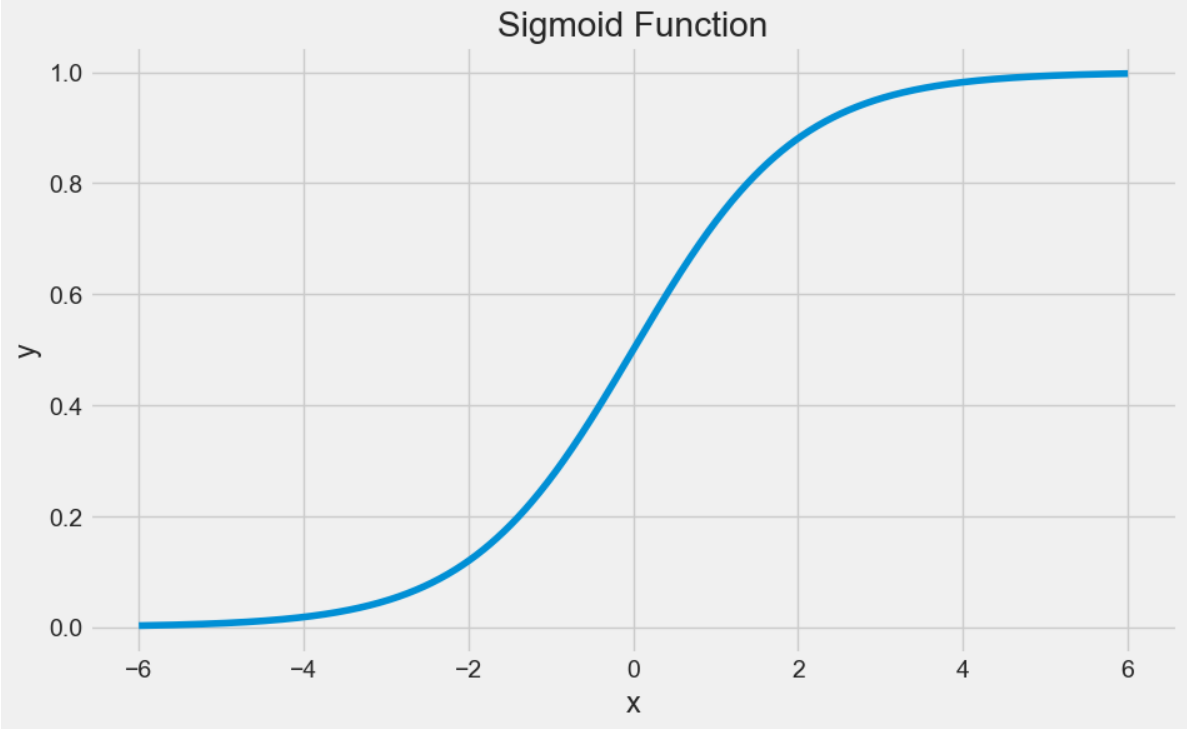
e is the base of the natural logarithms and x is value that you want to transform via the logistic function.

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

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

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

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



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

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

or

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

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

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

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

7.1.2 Learning the Logistic Regression Model

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

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

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

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

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

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

7.1.3 Preparing Data for Logistic Regression

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

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

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

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

7.2 Logistic Regression: Scikit-learn vs Statsmodels

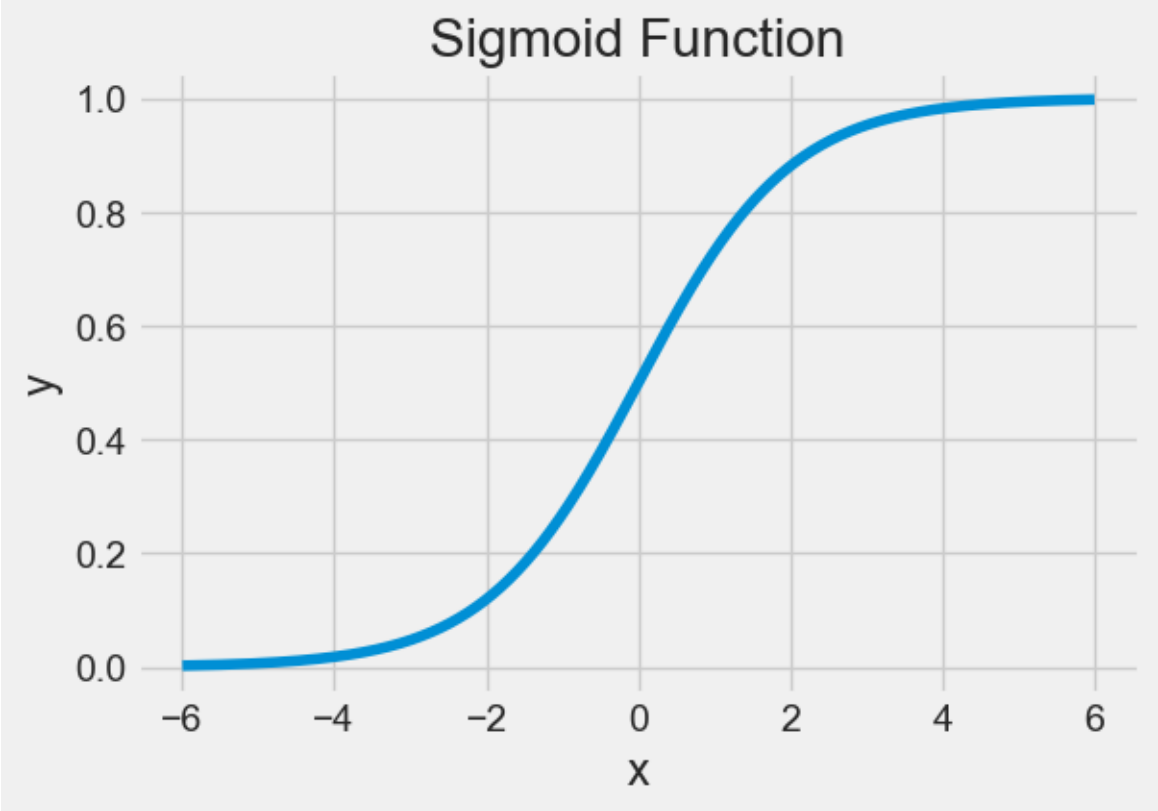
Python gives us two ways to do logistic regression. Statsmodels offers modeling from the perspective of statistics. Scikit-learn offers some of the same models from the perspective of machine learning.

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

7.3 Training a logistic regression model

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

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



7.4 Logistic Regression: Scikit-learn vs Statsmodels

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

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

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

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

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

```
train.head()
```

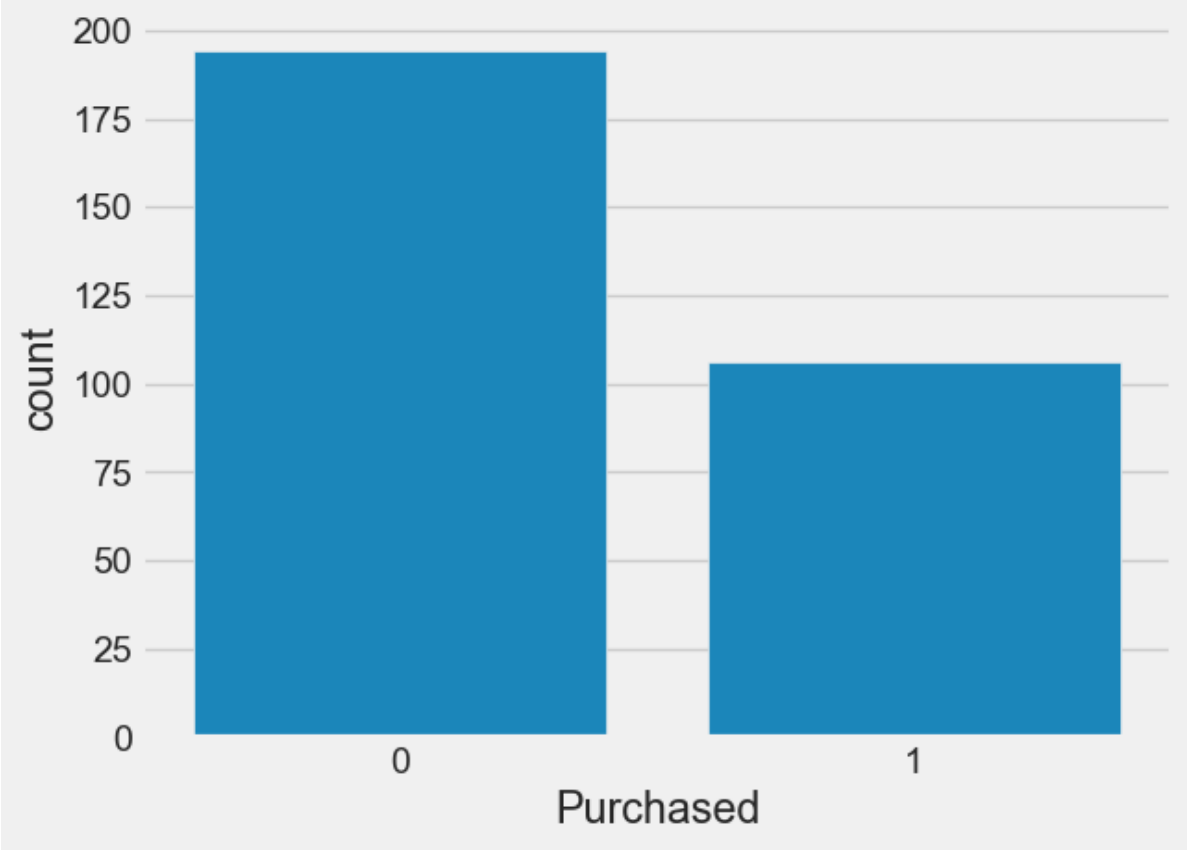
	User ID	Gender	Age	EstimatedSalary	Purchased
0	15755018	Male	36	33000	0
1	15697020	Female	39	61000	0
2	15796351	Male	36	118000	1
3	15665760	Male	39	122000	1
4	15794661	Female	26	118000	0

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

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

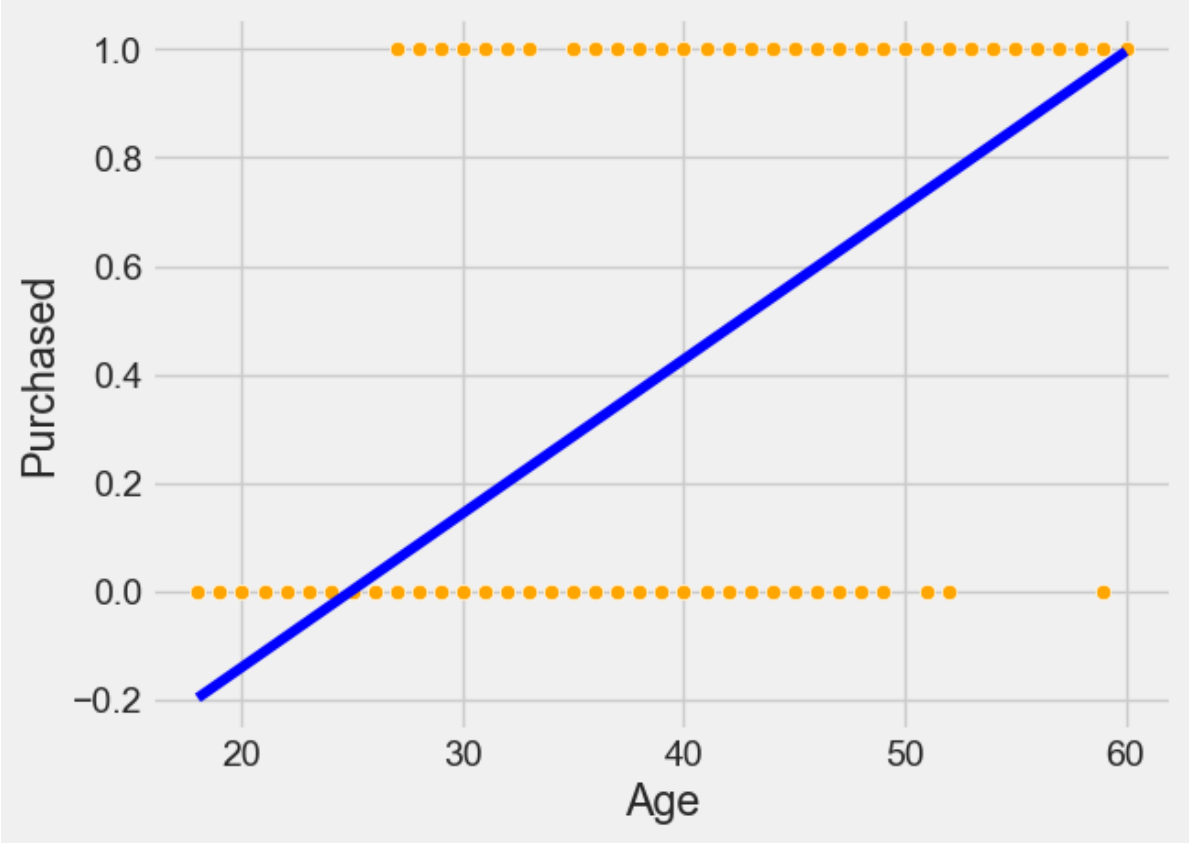
```
Purchased
0    194
1    106
Name: count, dtype: int64
```

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



7.4.2 Fitting a linear regression

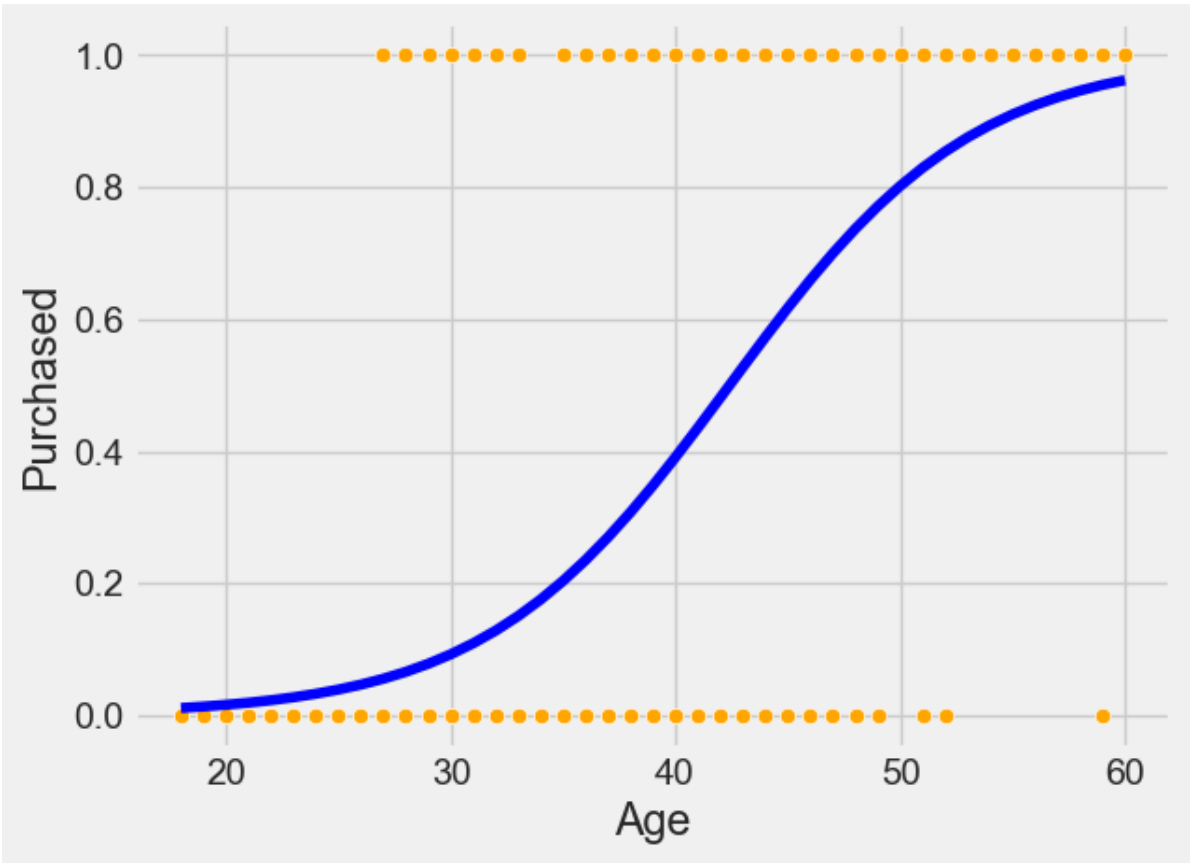
```
sns.scatterplot(x = 'Age', y = 'Purchased', data = train, color = 'orange') #Visualizing data
lm = sm.ols(formula = 'Purchased~Age', data = train).fit() #Developing linear regression mode
sns.lineplot(x = 'Age', y= lm.predict(train), data = train, color = 'blue') #Visualizing mode
```



7.4.3 Logistic Regression with Statsmodel

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

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




```
logit_model.summary()
```

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

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

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

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

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

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

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

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

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

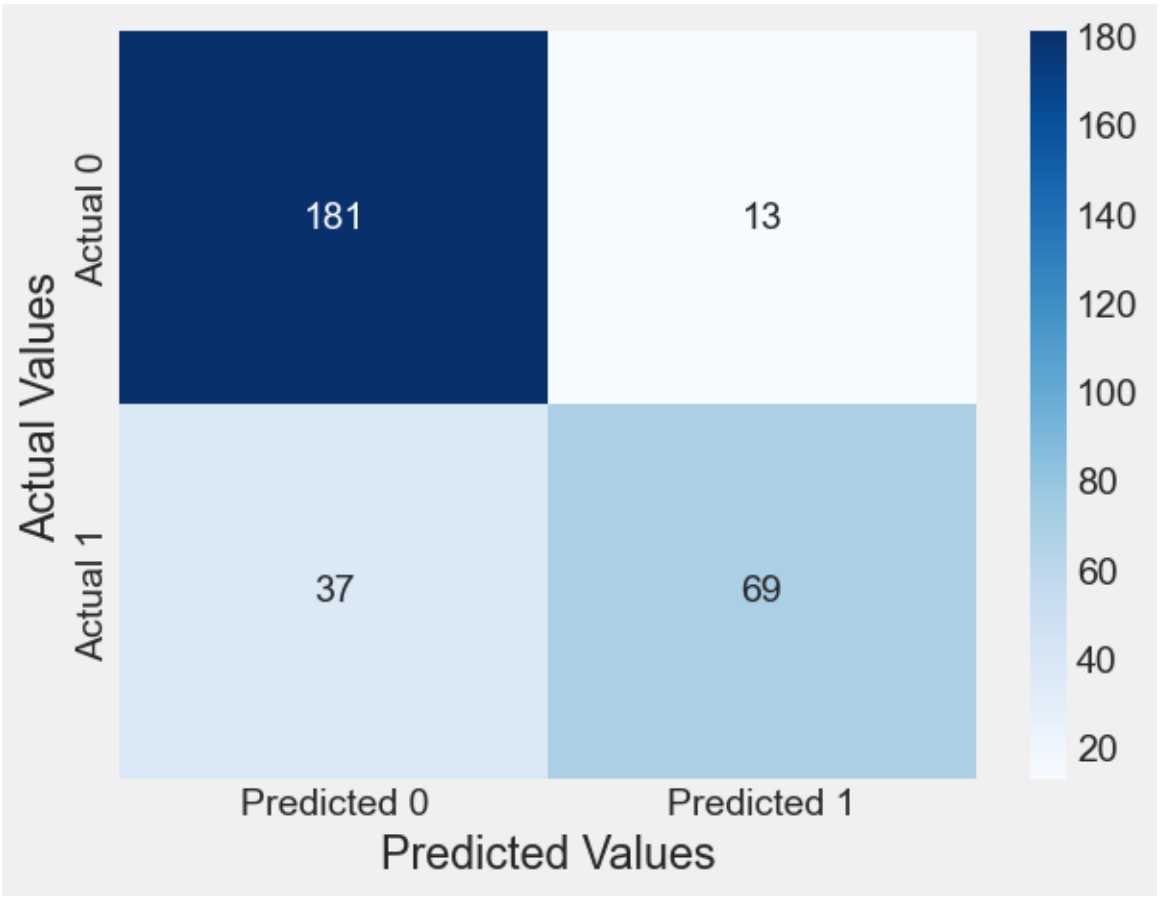
```
0      0
1      0
2      0
3      0
```

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

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

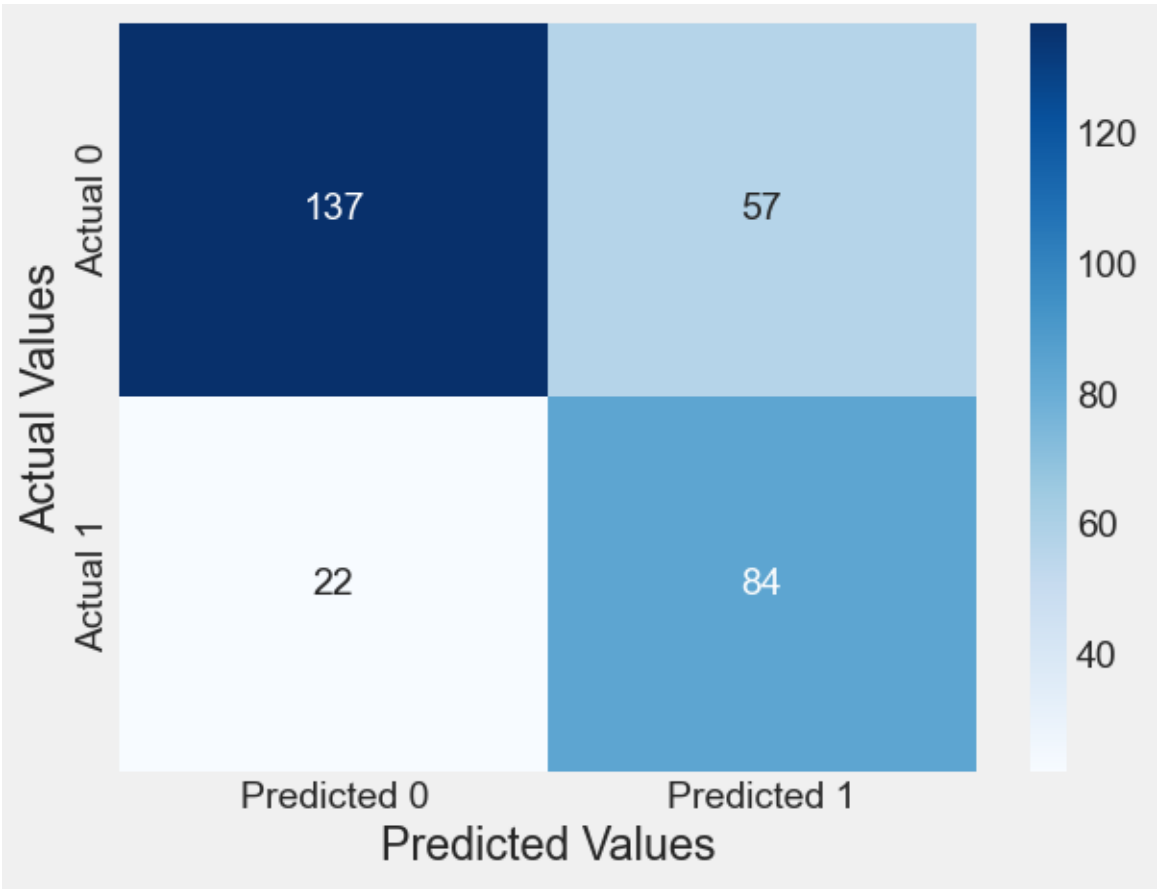
```
cm = confusion_matrix_train(logit_model)
```

```
Classification accuracy = 83.3%
```



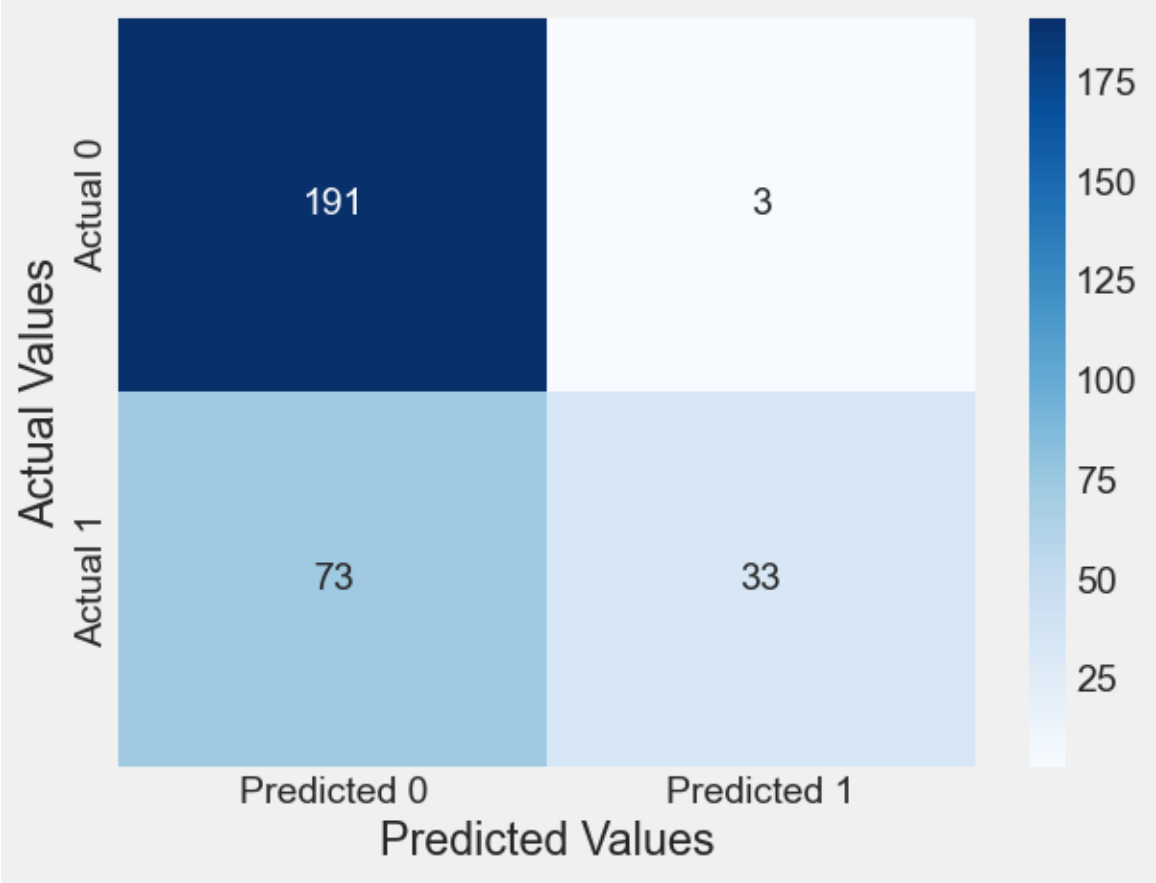

```
# change the cutoff to 0.3
cm = confusion_matrix_train(logit_model, 0.3)
```

```
Classification accuracy = 73.7%
```




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

```
Classification accuracy = 74.7%
```



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

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

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

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

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

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

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

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

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

7.4.4 Logistic Regression with Sklearn

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

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

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

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

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

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

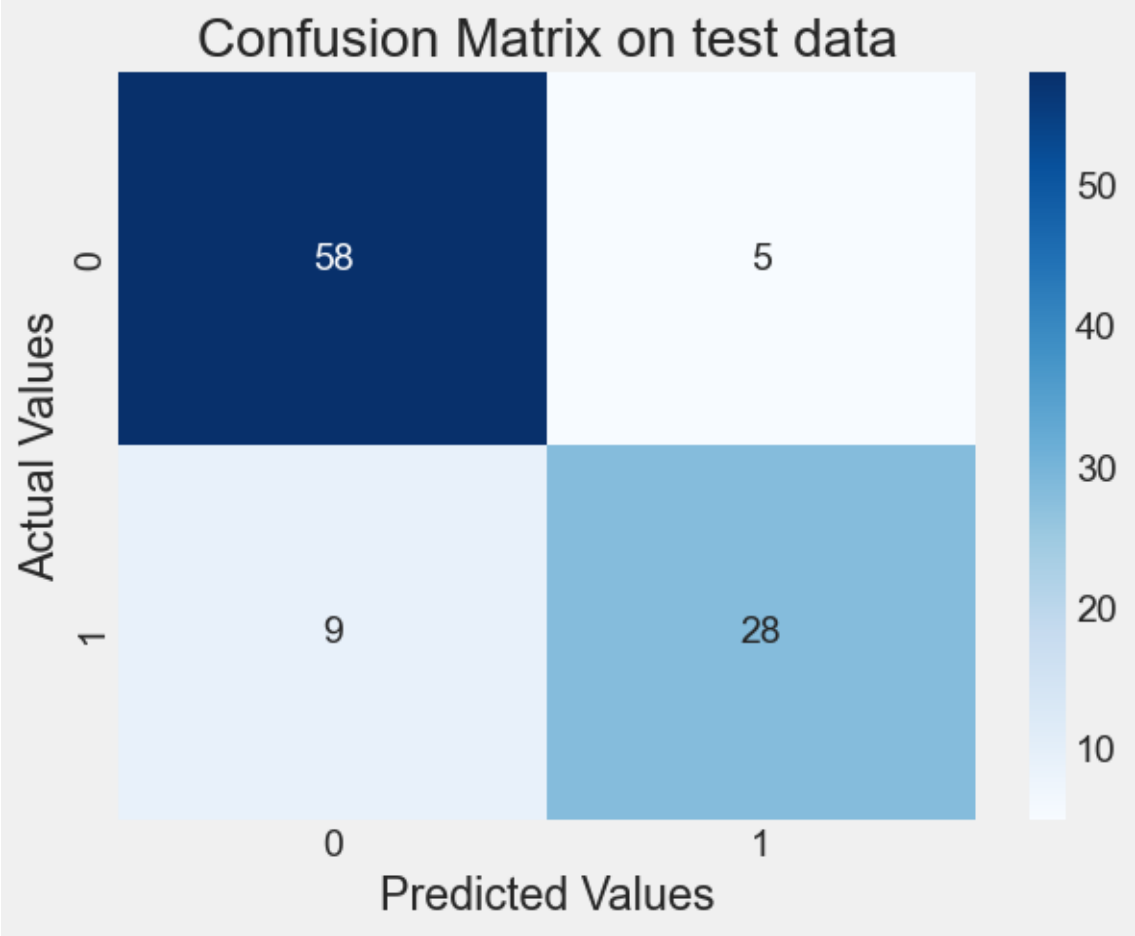
```
array([0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0,
       1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1,
    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

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

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

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

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




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

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

7.4.5 Changing the default threshold

```
new_threshold = 0.3
```

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

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

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

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

7.5 Performance Measurement

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

7.5.1 Precision-recall

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

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

==> `70.13%`

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

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

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

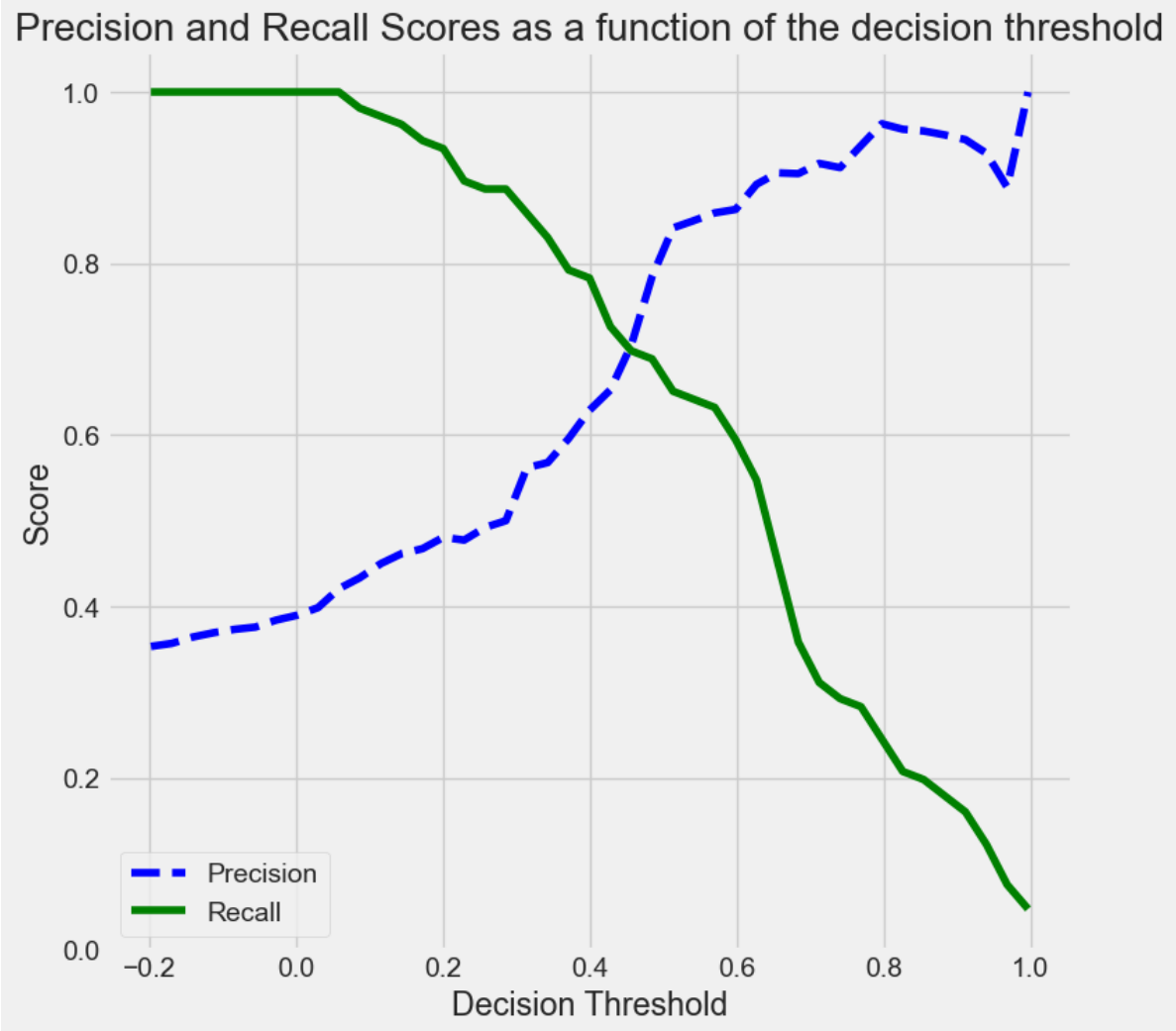
Visualize the precision-recall curve for the model `logit_model_diabetes`.

```
train
```

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

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

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



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

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

Hint: Look at the documentation for `precision_recall_curve`.

7.5.2 The Receiver Operating Characteristics (ROC) Curve

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

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

High Threshold:

- High specificity
- Low sensitivity

Low Threshold

- Low specificity
- High sensitivity

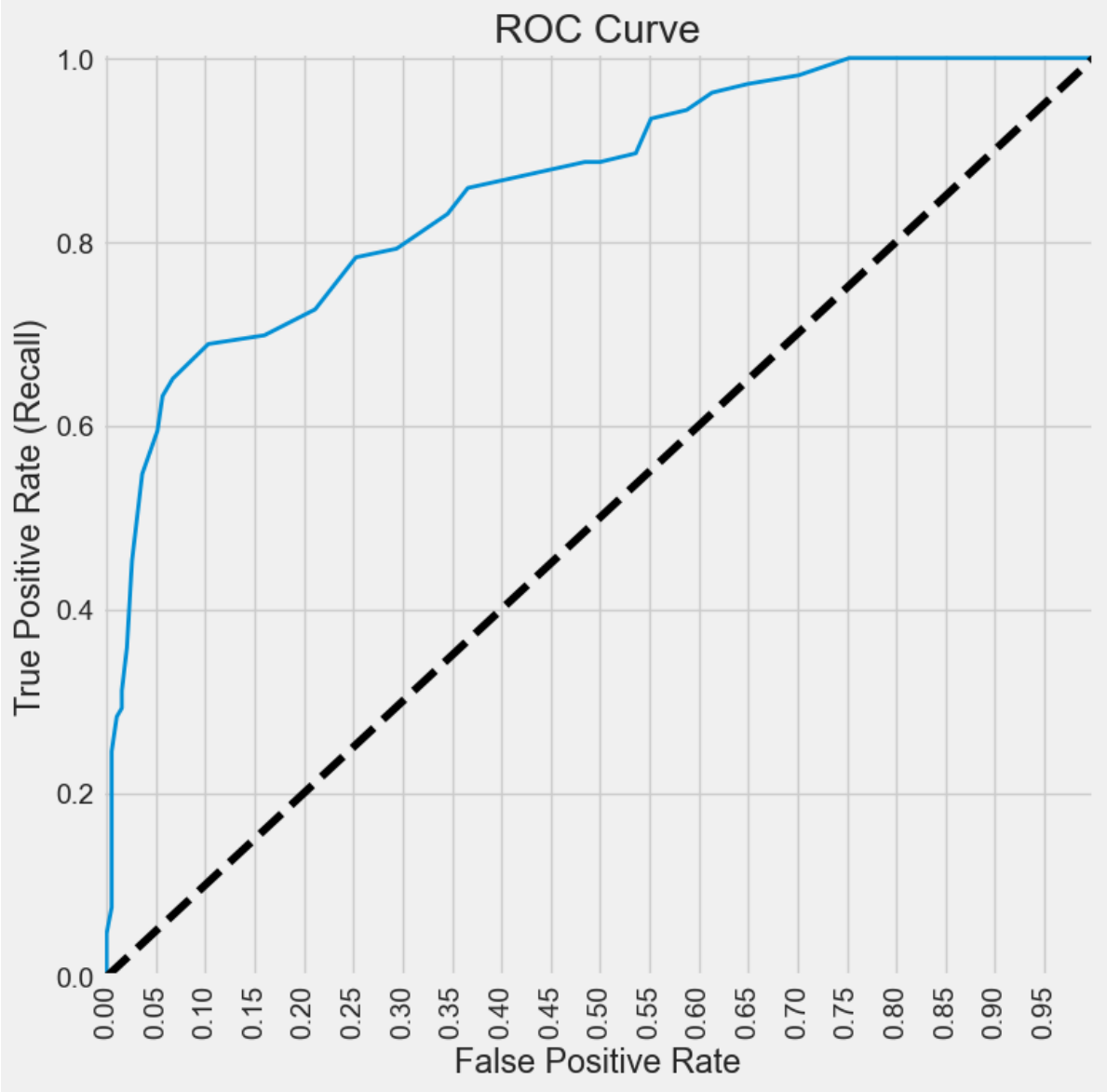
The area under ROC is called *Area Under the Curve(AUC)*. AUC gives the rate of successful classification by the logistic model. To get a more in-depth idea of what a ROC-AUC curve is and how is it calculated, here is a good blog link.

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

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

```
y=train.Purchased
ypred = lm.predict(train)
fpr, tpr, auc_thresholds = roc_curve(y, ypred)
print(auc(fpr, tpr))# AUC of ROC
def plot_roc_curve(fpr, tpr, label=None):
    plt.figure(figsize=(8,8))
    plt.title('ROC Curve')
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.005, 1, 0, 1.005])
    plt.xticks(np.arange(0,1, 0.05), rotation=90)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate (Recall)")

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

```
0.8593901964598327
```



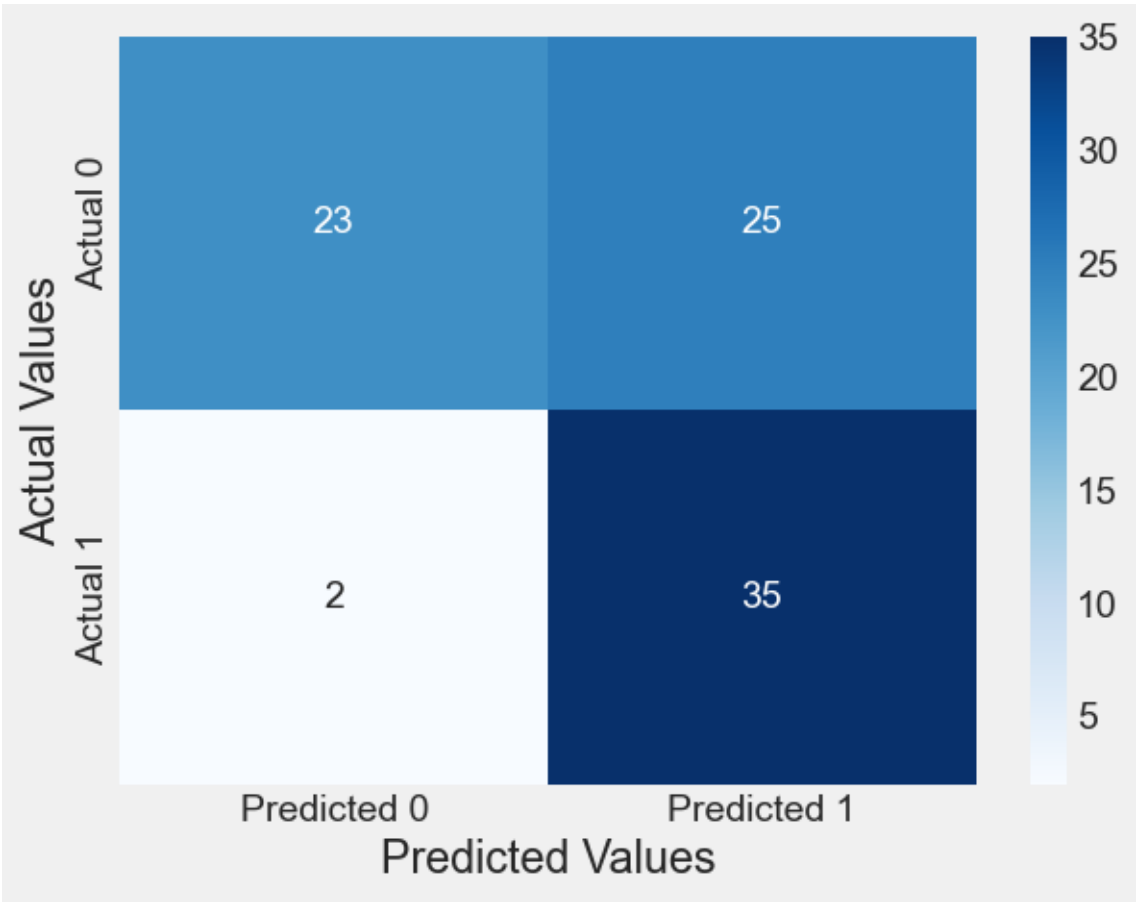
Q: How are the values of the `auc_thresholds` chosen to make the ROC curve? Why does it look like a step function?

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

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

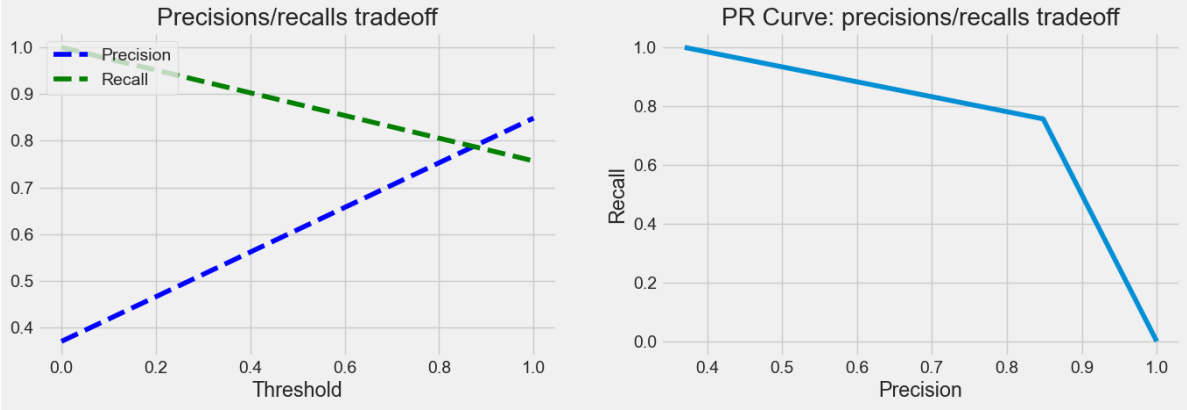
```
confusion_matrix_data(test,test.Purchased,lm,0.3)
```

```
Classification accuracy = 68.2%
Precision = 58.3%
TPR or Recall = 94.6%
FNR = 5.4%
FPR = 52.1%
ROC-AUC = 89.4%
```


Actual Values
Actual 0
Actual 1
23
25
2
35
Predicted 0
Predicted 1
Predicted Values
35
30
25
20
15
10
5

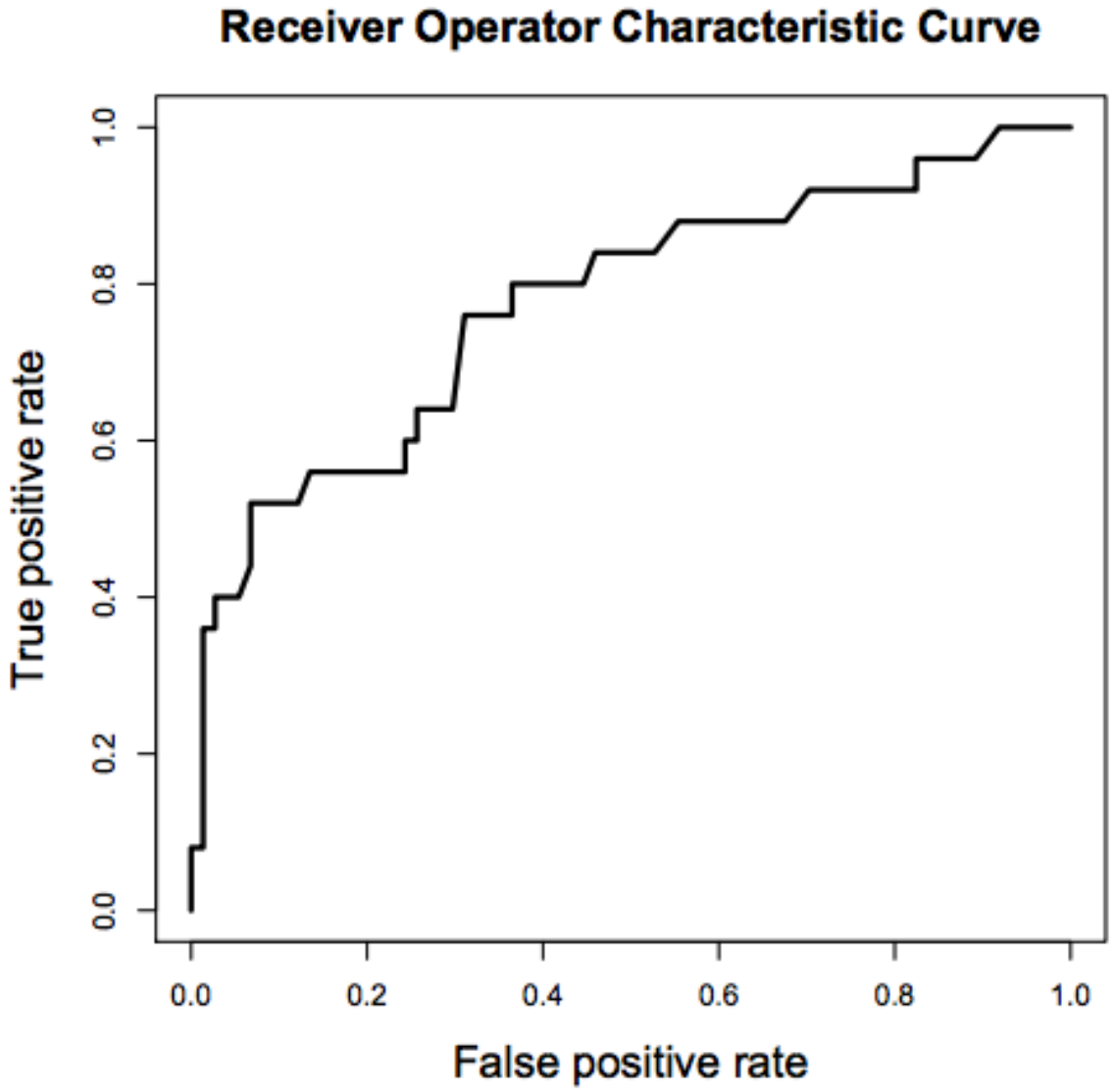
8 Precision/Recall Tradeoff

```
from sklearn.metrics import precision_recall_curve


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


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


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

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



8.1 The Receiver Operating Characteristics (ROC) Curve

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



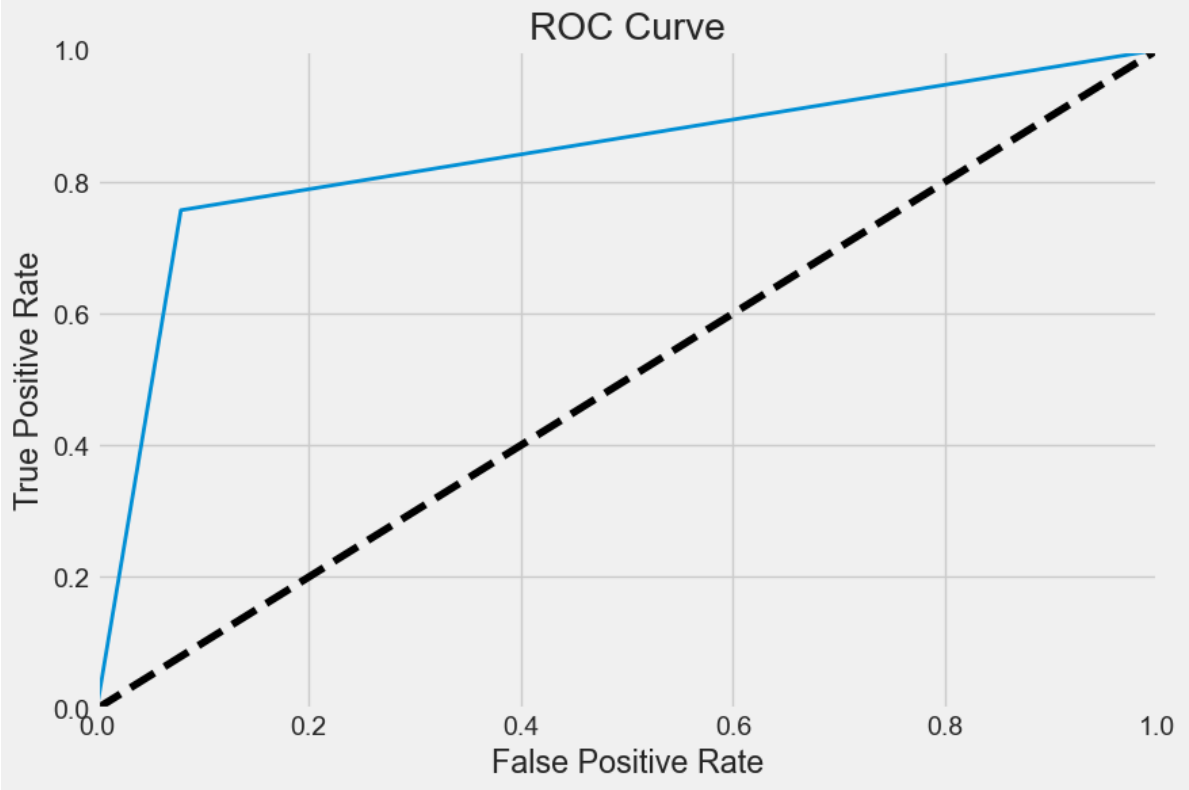
High Threshold: * High specificity * Low sensitivity

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 link

```
from sklearn.metrics import roc_curve
```

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], "k--")
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')

fpr, tpr, thresholds = roc_curve(y_test, y_pred_test)
plt.figure(figsize=(9,6));
plot_roc_curve(fpr, tpr)
plt.show();
```




```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_test, y_pred_test)
```

```
0.8386958386958387
```

9 Logistic regression: Others

```
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

sns.set()
```

9.1 Decision Boundary in Classification

```
# Importing the dataset
dataset = pd.read_csv('datasets/apples_and_oranges.csv')
dataset.head()
```

	Weight	Size	Class
0	69	4.39	orange
1	69	4.21	orange
2	65	4.09	orange
3	72	5.85	apple
4	67	4.70	orange

```
# No. of apples and oranges
dataset['Class'].value_counts()
```

```
Class
orange    20
apple     20
Name: count, dtype: int64
```

9.1.1 Encoding Target

```
le = LabelEncoder()
dataset['Class'] = le.fit_transform(dataset['Class'])
le.classes_
```

```
array(['apple', 'orange'], dtype=object)
```

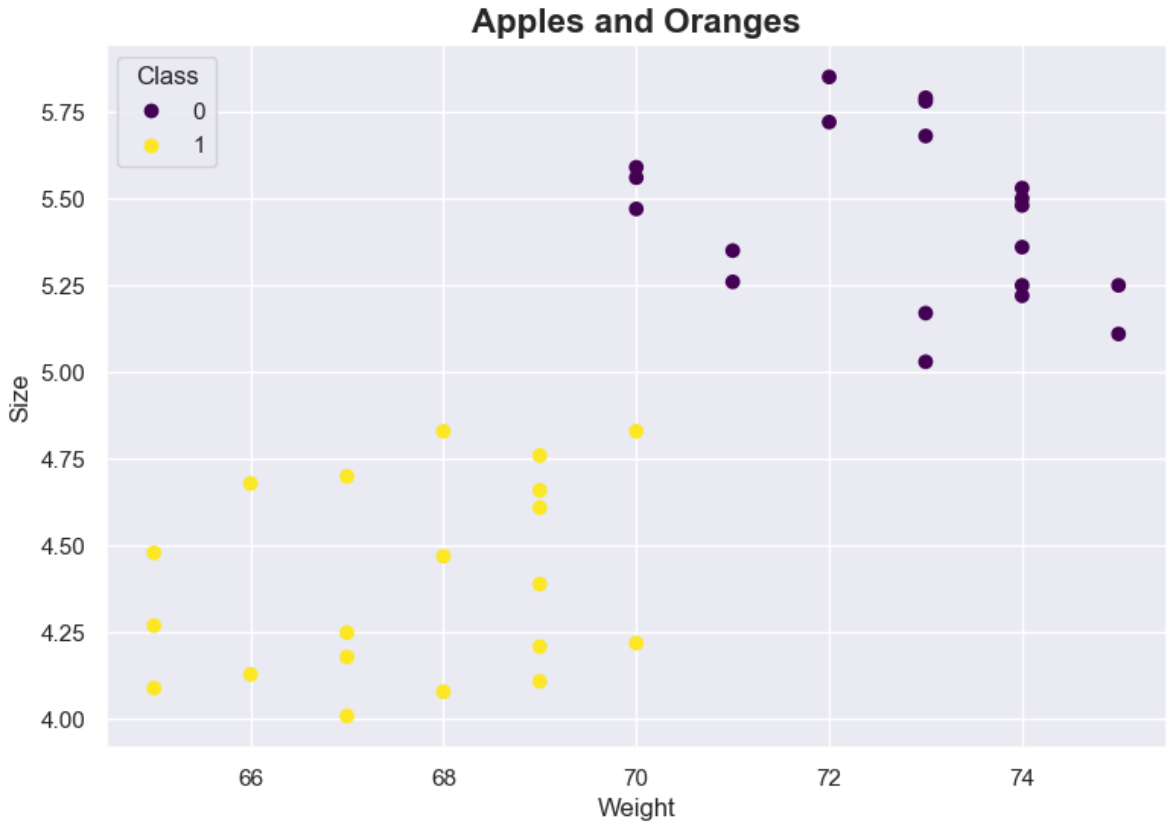
This implies that, * 0 represents Apple * 1 represents Orange

```
dataset.head()
```

	Weight	Size	Class
0	69	4.39	1
1	69	4.21	1
2	65	4.09	1
3	72	5.85	0
4	67	4.70	1

9.1.2 Plotting the dataset

```
plt.figure(figsize=(9,6))
plt.title('Apples and Oranges', fontweight='bold', fontsize=16)
plt.xlabel('Weight')
plt.ylabel('Size')
scatter = plt.scatter(dataset['Weight'], dataset['Size'], c=dataset['Class'], cmap='viridis')
plt.legend(*scatter.legend_elements(),
           loc = 'upper left',
           title = 'Class');
```



We can observe that oranges have lower weight and size compared to apples. Further by drawing a straight line between these two groups of data points, we can clearly distinguish between apples and oranges.

9.1.3 Building a Logistic Regression model to distinguish apples and oranges

As we can clearly distinguish between apples and oranges using a straight line decision boundary, we can choose the hypothesis $y = a0 + a1$ x1 + a2 * x2* for Logistic Regression where, *a0*, *a1*, *a2* are the fitting parameters *x1* is Weight *x2* is Size

```
# Defining target and features
y = dataset['Class']
x = dataset.drop(columns=['Class'])
```

```
# Creating object of LogisticRegression class
log_reg = LogisticRegression()
```

```
# Fitting parameters
log_reg.fit(x,y)
```

```
LogisticRegression()
```

```
# Intercept - a0
log_reg.intercept_
```

```
array([106.60287324])
```

```
# Coefficients - a1, a2 respectively
log_reg.coef_
```

```
array([[-1.42833694, -1.31285258]])
```

```
# Predicting labels for the given dataset
label_predictions = log_reg.predict(x)
label_predictions
```

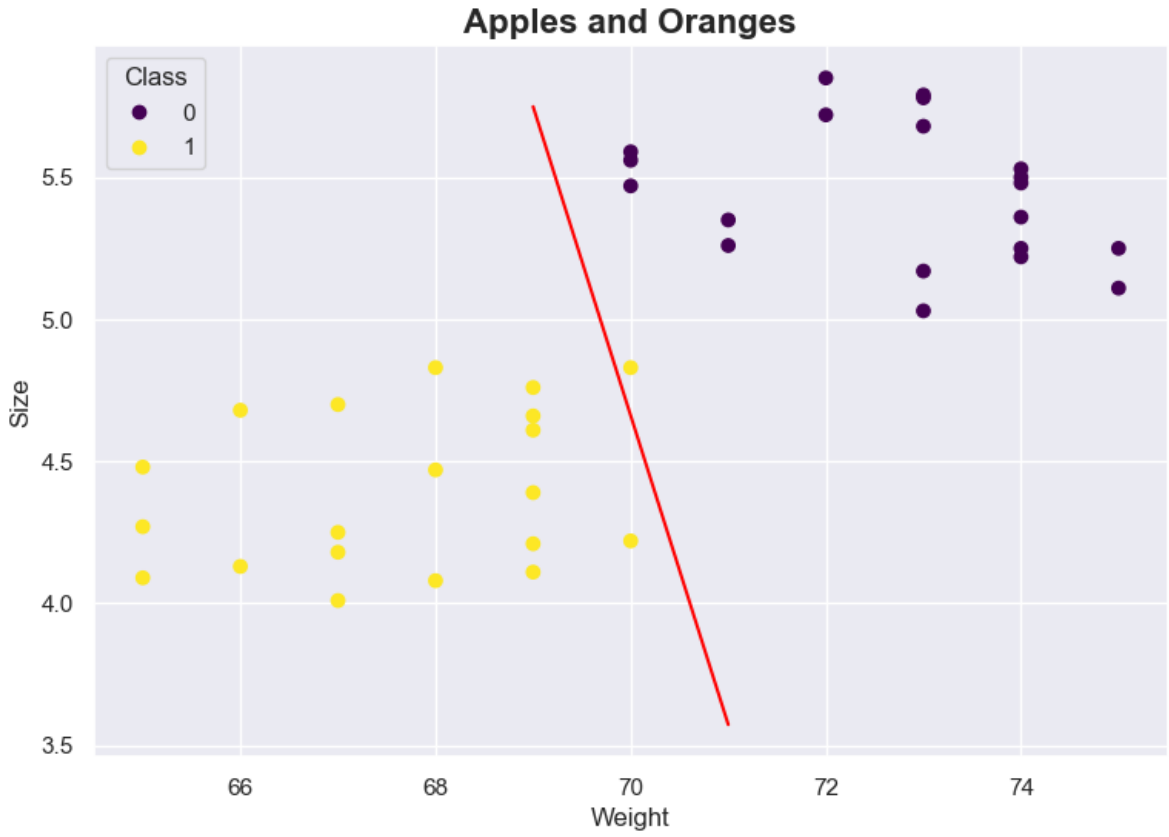
```
array([1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1,
       1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0])
```

9.1.4 Linear Decision Boundary with naive features

```
# Parameter values
a0 = log_reg.intercept_[0]
a1 = log_reg.coef_[0][0]
a2 = log_reg.coef_[0][1]
```

```
# Defining x1 and x2 values for decision boundary
x1 = np.array([69, 71])
x2 = (0.5 -a0 - (a1 * x1)) / a2
```

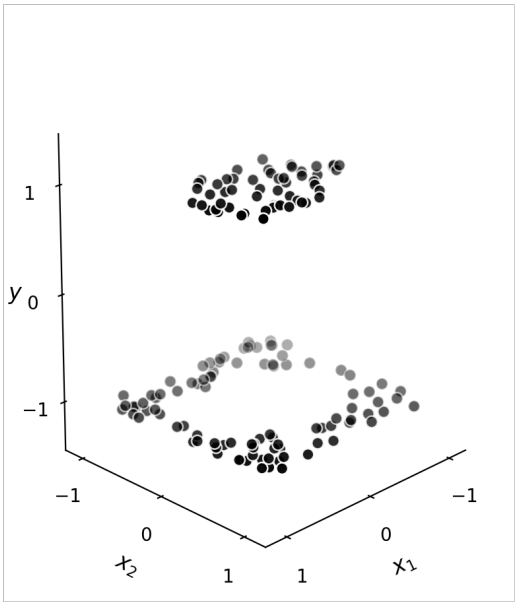
```
# Plotting the decision boundary
plt.figure(figsize=(9,6))
plt.title('Apples and Oranges', fontweight='bold', fontsize=16)
plt.xlabel('Weight')
plt.ylabel('Size')
scatter = plt.scatter(dataset['Weight'], dataset['Size'], c=dataset['Class'], cmap='viridis')
plt.legend(*scatter.legend_elements(),
           loc = 'upper left',
           title = 'Class')
plt.plot(x1, x2, color='red', label='Decision Boundary')
plt.show()
```

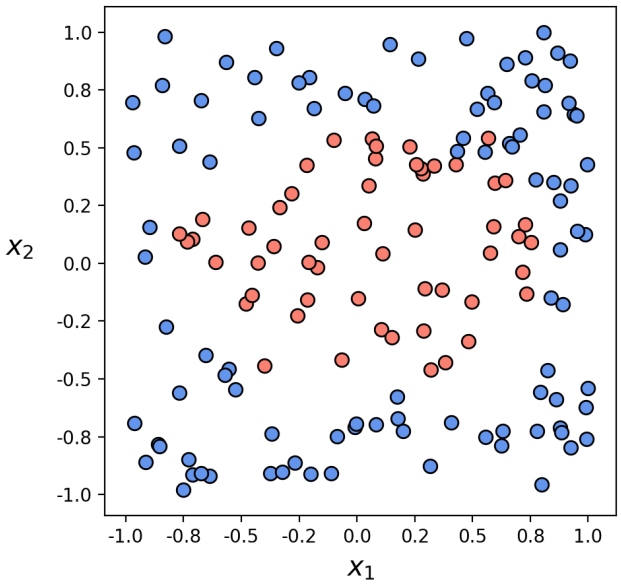


In this problem, we have just two features x1 and x2, if we use just those as they are we will end up with a straight line which divides our 2D plane into two half-planes.

9.2 Non-linear Decision Boundary

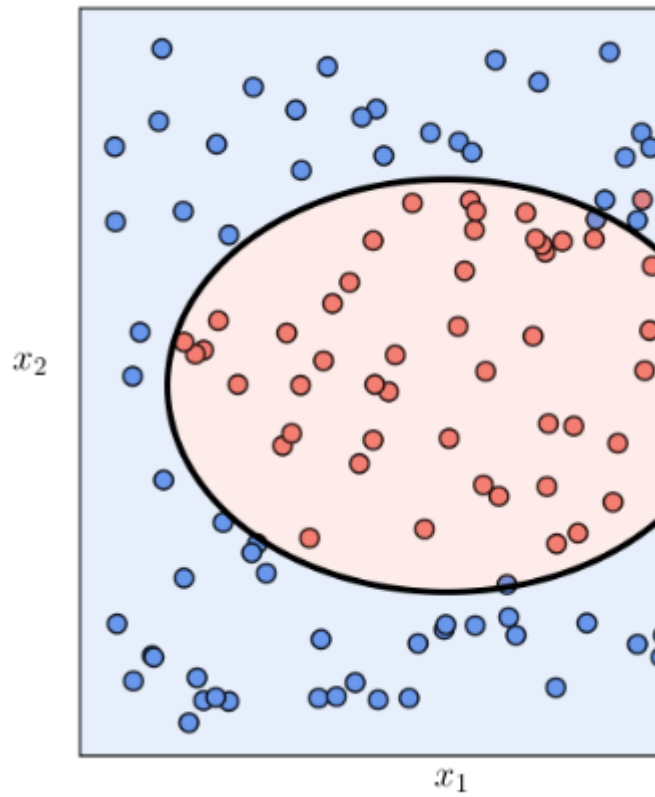
In some occasions, we want to have a more complex boundary, and we can achieve this by transforming our features. For instance, when confronted with a training data distribution as illus-





trated below,
it becomes imperative to generate polynomial features

$$(x_1^2, x_2^2)$$



in order to enhance the delineation between the two classes.

Let's delve into a concrete example below to illustrate this concept.For the purpose of illustrating the decision boundary, I chose not to split the data into training and test sets.

```
#Load our Dataset for Logistic Regression
components = pd.read_csv('datasets/ex2data2.txt', header=None, names = ['feature 1', 'feature
components.head()
```

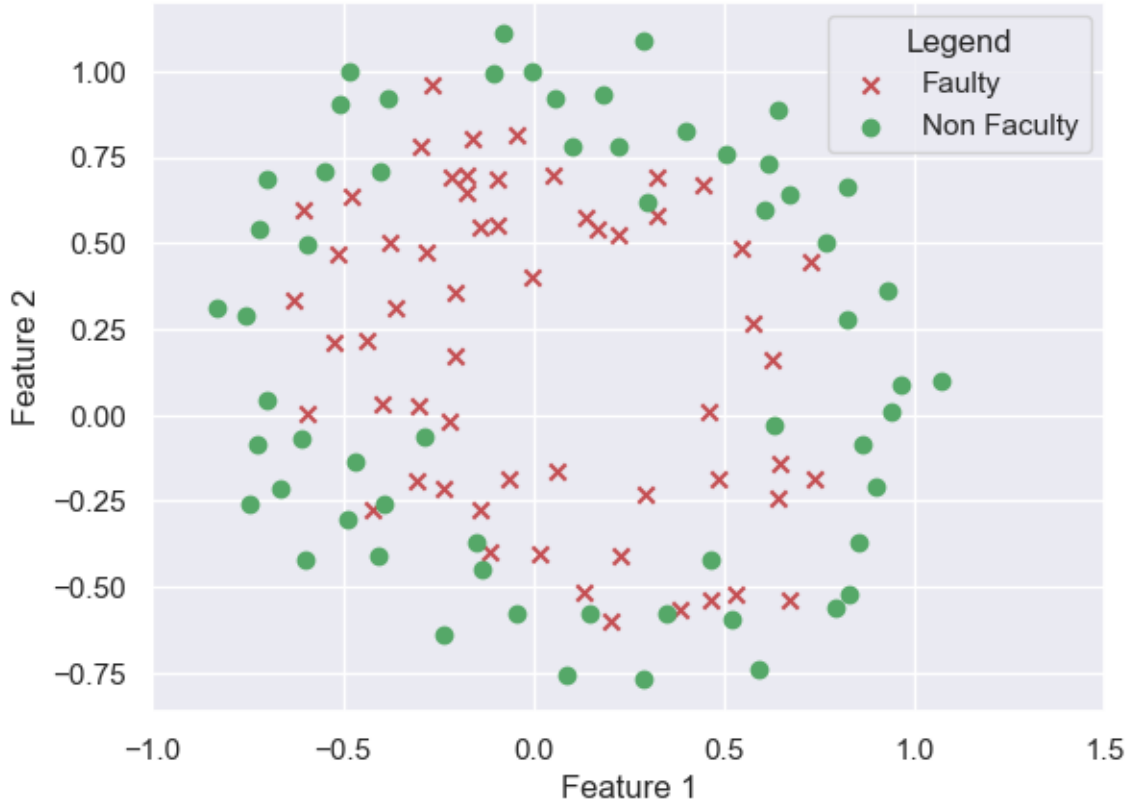
	feature 1	feature 2	faulty
0	0.051267	0.69956	1
1	-0.092742	0.68494	1
2	-0.213710	0.69225	1
3	-0.375000	0.50219	1
4	-0.513250	0.46564	1

```
# check the balance of the dataset
components['faulty'].value_counts()
```

```
faulty
0    60
1    58
Name: count, dtype: int64
```

```
# get positive and negative samples for plotting
pos = components['faulty'] == 1
neg = components['faulty'] == 0
```

```
# Visualize Data
fig, axes = plt.subplots();
axes.set_xlabel('Feature 1')
axes.set_ylabel('Feature 2')
axes.scatter(components.loc[pos, 'feature 1'], components.loc[pos, 'feature 2'], color = 'r'
axes.scatter(components.loc[neg, 'feature 1'], components.loc[neg, 'feature 2'], color = 'g'
axes.legend(title='Legend', loc = 'best' )
axes.set_xlim(-1,1.5)
axes.set_xlim(-1,1.5)
```



As we can see that the positive and negative examples are not linearly seperable. So we have to add additional higher order polynomial features.

```
# define function to map higher order polynomial features
def mapFeature(X1, X2, degree):
    res = np.ones(X1.shape[0])
    for i in range(1,degree + 1):
        for j in range(0,i + 1):
            res = np.column_stack((res, (X1 ** (i-j)) * (X2 ** j)))

    return res
```

```
# Get the features
X = components.iloc[:, :2]
```

```
degree = 2
```

```
X_poly = mapFeature(X.iloc[:, 0], X.iloc[:, 1], degree)
```

```
# Get the target variable
y = components.iloc[:, 2]
```

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

```
def costFunc(theta, X, y):
    m = y.shape[0]
    z = X.dot(theta)
    h = sigmoid(z)
    term1 = y * np.log(h)
    term2 = (1- y) * np.log(1 - h)
    J = -np.sum(term1 + term2, axis = 0) / m
    return J
```

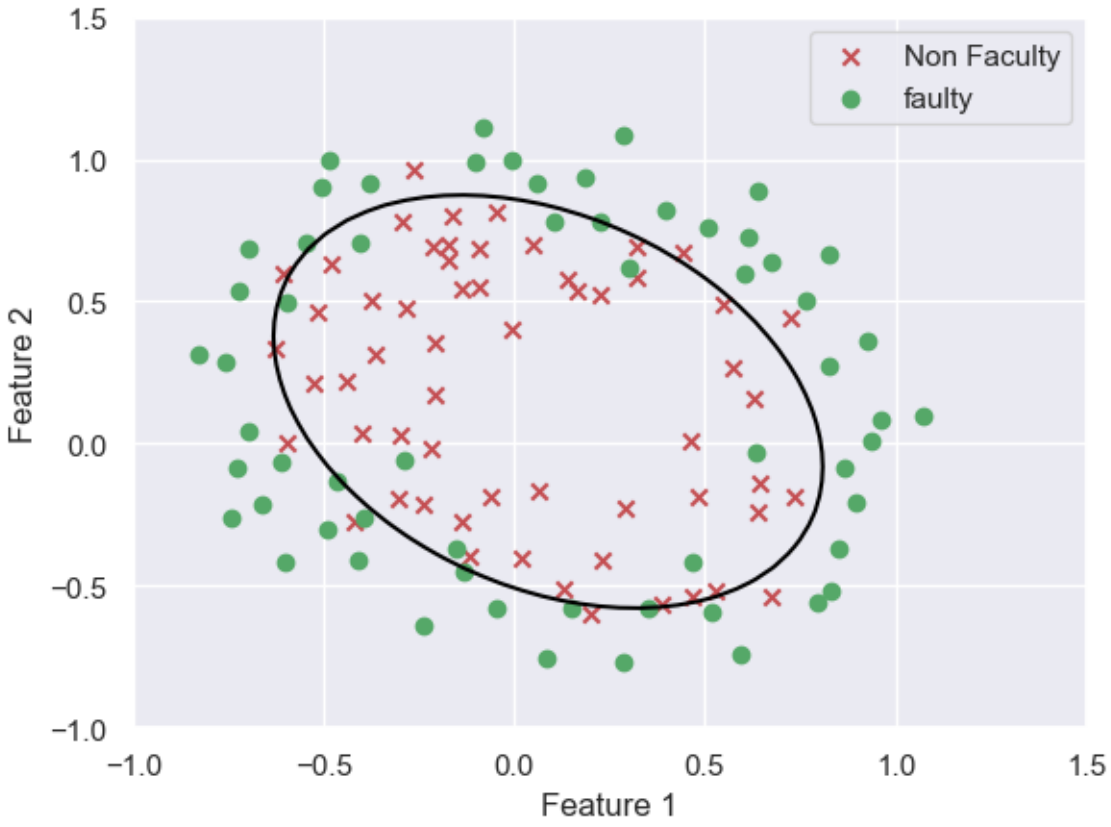
```
# Set initial values for our parameters
initial_theta = np.zeros(X_poly.shape[1]).reshape(X_poly.shape[1], 1)
```

```
# Now call the optimization routine
#NOTE: This automatically picks the learning rate
from scipy.optimize import minimize
res = minimize(costFunc, initial_theta.flatten(), args=(X_poly, y))
```

```
# our optimizated coefficients
theta = res.x
```

```
# define a function to plot the decision boundary
def plotDecisionBoundary(theta,degree, axes):
    u = np.linspace(-1, 1.5, 50)
    v = np.linspace(-1, 1.5, 50)
    U,V = np.meshgrid(u,v)
    # convert U, V to vectors for calculating additional features
    # using vectorized implementation
    U = np.ravel(U)
    V = np.ravel(V)
    Z = np.zeros((len(u) * len(v)))

    X_poly = mapFeature(U, V, degree)
    Z = X_poly.dot(theta)

    # reshape U, V, Z back to matrix
    U = U.reshape((len(u), len(v)))
    V = V.reshape((len(u), len(v)))
    Z = Z.reshape((len(u), len(v)))

    cs = axes.contour(U,V,Z,levels=[0],cmap= "Greys_r")
    axes.legend(labels=['Non Faculty', 'faulty', 'Decision Boundary'])
    return cs
```

```
# Plot Decision boundary
fig, axes = plt.subplots();
axes.set_xlabel('Feature 1')
axes.set_ylabel('Feature 2')
axes.scatter(components.loc[pos, 'feature 1'], components.loc[pos, 'feature 2'], color = 'r'
axes.scatter(components.loc[neg, 'feature 1'], components.loc[neg, 'feature 2'], color = 'g'
#axes.legend(title='Legend', loc = 'best' )

plotDecisionBoundary(theta, degree, axes);
```

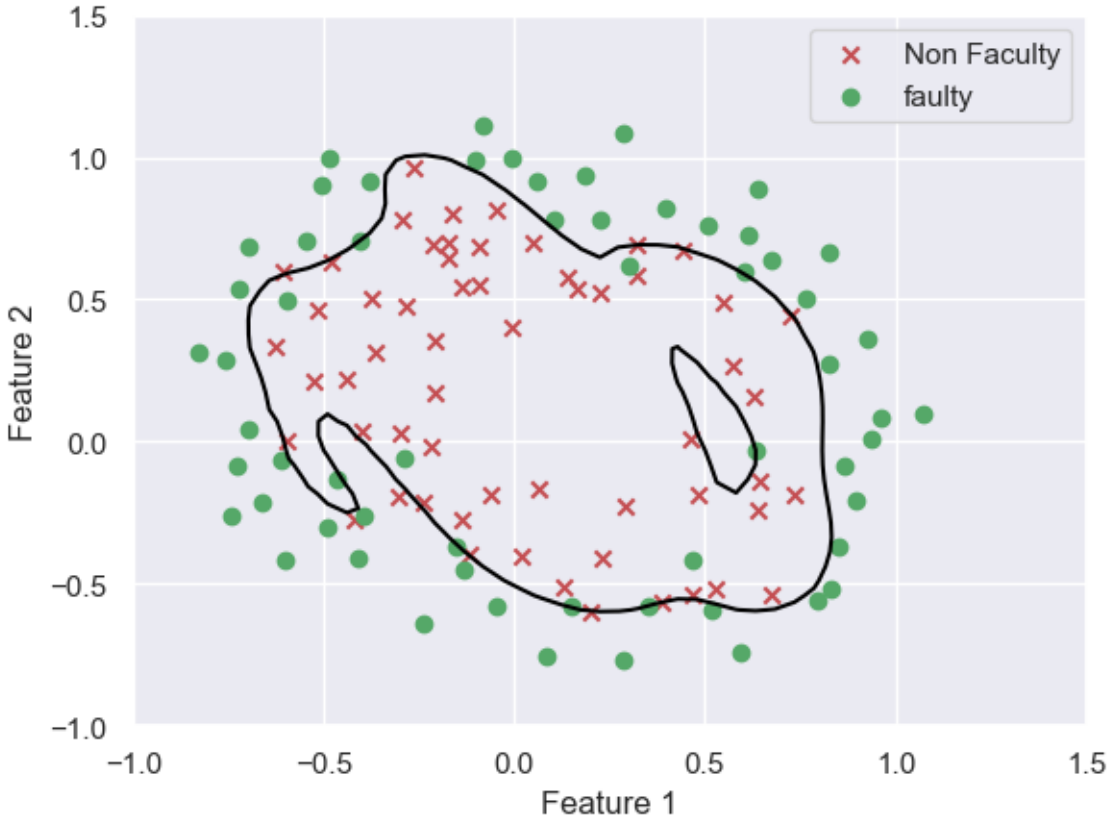


of course, you can increase the degree of the polynomial you want to fit, but the overfitting could become a problem

```
# set degree = 6
degree = 6
# map features to the degree
X_poly = mapFeature(X.iloc[:, 0], X.iloc[:, 1], degree)
# set initial parameters
initial_theta = np.zeros(X_poly.shape[1]).reshape(X_poly.shape[1], 1)
```

```
# Run the optimization function
res = minimize(costFunc, initial_theta.flatten(), args=(X_poly, y))
theta = res.x.reshape(res.x.shape[0], 1)

# Plot Decision boundary
fig, axes = plt.subplots()
axes.set_xlabel('Feature 1')
axes.set_ylabel('Feature 2')
```

```
axes.scatter(components.loc[pos, 'feature 1'], components.loc[pos, 'feature 2'], color='r', m
axes.scatter(components.loc[neg, 'feature 1'], components.loc[neg, 'feature 2'], color='g', m
#axes.legend(title='Legend', loc='best')

plotDecisionBoundary(theta, degree, axes)
```



As we can see the model tries pretty hard to capture every single example perfectly and overfits the data. This kind of model has overfitting issue. i.e The model has not pre-conceived notion about the seperation of the positive and negative examples and pretty much can fit any kind of data. Such model will fail in predicting the correct classification when it sees new examples.

One of techiques is to use regularization, which we will cover later. The idea is to penalize the algorithm when it tries to overfit by adding a regularization term to the cost function.

The New Cost function with the regularization is specified as

$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y_i log(h_\theta(z_i) - (1-y_i) log(1-h_\theta(z_i))] + \frac{\lambda}{2m}\sum_{j=1}^{n}[\theta_j^2]$

where λ = regularization factor n = number of features. (NOTE: The regularization term does include the intercept term θ_0

9.2.1 By adding Polynomial Features

```
components
```

	feature 1	feature 2	faulty
0	0.051267	0.699560	1
1	-0.092742	0.684940	1
2	-0.213710	0.692250	1
3	-0.375000	0.502190	1
4	-0.513250	0.465640	1
...	...	...	...
113	-0.720620	0.538740	0
114	-0.593890	0.494880	0
115	-0.484450	0.999270	0
116	-0.006336	0.999270	0
117	0.632650	-0.030612	0

```
X = components[['feature 1', 'feature 2']]
y = components['faulty']
```

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)
```

```
feature_names = poly.get_feature_names_out()
print(feature_names)
```

```
['1' 'feature 1' 'feature 2' 'feature 1^2' 'feature 1 feature 2'
 'feature 2^2']
```

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_poly, y)
```

```
LogisticRegression()
```

```
label_predictions = model.predict(X_poly)
```

9.2.1.1 Accuracy Score

```
accuracy_score(y, label_predictions)
```

```
0.8135593220338984
```

9.2.1.2 Confusion Matrix

```
cm = confusion_matrix(y, label_predictions)
cm
```

```
array([[47, 13],
       [ 9, 49]], dtype=int64)
```

9.2.2 By Transforming Continuous Variable

Variable transformation is an important technique to create robust models using logistic regression. Because the predictors are linear in the log of the odds, it is often helpful to transform the continuous variables to create a more linear relationship. To determine the best transformation of a continuous variable, a univariate plot is very helpful. Remember the nice univariate plot of Y variable against X variable in linear regression? This is not easily attained, because Y is dichotomous in logistic regression.

There are different recommended solutions. Among them, * One is to create several variations (in forms of squared, cubed, or logged transformations etc.). * Another solution is to break some continuous variables into segments and treat them as categorical variables. This may work well to pick up nonlinear trends. The biggest drawback is that it loses the benefit of the linear trend relationship in the curve1. It also may lead to over fitting.

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

```
train.head()
```

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

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300 entries, 0 to 299
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   User ID          300 non-null    int64
 1   Gender           300 non-null    object
 2   Age              300 non-null    int64
 3   EstimatedSalary  300 non-null    int64
 4   Purchased        300 non-null    int64
dtypes: int64(4), object(1)
memory usage: 11.8+ KB
```

```
train.Gender.value_counts()
```

```
Gender
Female    151
Male      149
Name: count, dtype: int64
```

```
tmp_1 = pd.get_dummies(train['Gender'], drop_first=True)
train = pd.concat([train, tmp_1], axis=1)
train.head()
```

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

	User ID	Gender	Age	EstimatedSalary	Purchased	Male
4	15794661	Female	26	118000	0	False

```
tmp_1 = pd.get_dummies(test['Gender'], drop_first=True)
test = pd.concat([test, tmp_1], axis=1)
test.head()
```

	User ID	Gender	Age	EstimatedSalary	Purchased	Male
0	15810944	Male	35	20000	0	True
1	15668575	Female	26	43000	0	False
2	15603246	Female	27	57000	0	False
3	15694829	Female	32	150000	1	False
4	15697686	Male	29	80000	0	True

```
# Separating features and target on training set
y_train = train.Purchased
X_train = train.drop(["Purchased", "Gender", "User ID"], axis = 1)
```

```
X_train
```

	Age	EstimatedSalary	Male
0	36	33000	True
1	39	61000	False
2	36	118000	True
3	39	122000	True
4	26	118000	False
...	...	...	...
295	48	96000	False
296	42	149000	True
297	28	79000	True
298	51	134000	False
299	33	28000	False

```
# Separating features and target on test set
y_test = test.Purchased
X_test = test.drop(["Purchased", "Gender", "User ID"], axis = 1)
X_test
```

	Age	EstimatedSalary	Male
0	35	20000	True
1	26	43000	False
2	27	57000	False
3	32	150000	False
4	29	80000	True
...	...	...	...
95	49	39000	False
96	47	34000	True
97	60	42000	True
98	39	59000	False
99	51	23000	True

```
from sklearn.linear_model import LogisticRegression
sklearn_model = LogisticRegression()
sklearn_model.fit(X_train, y_train)
```

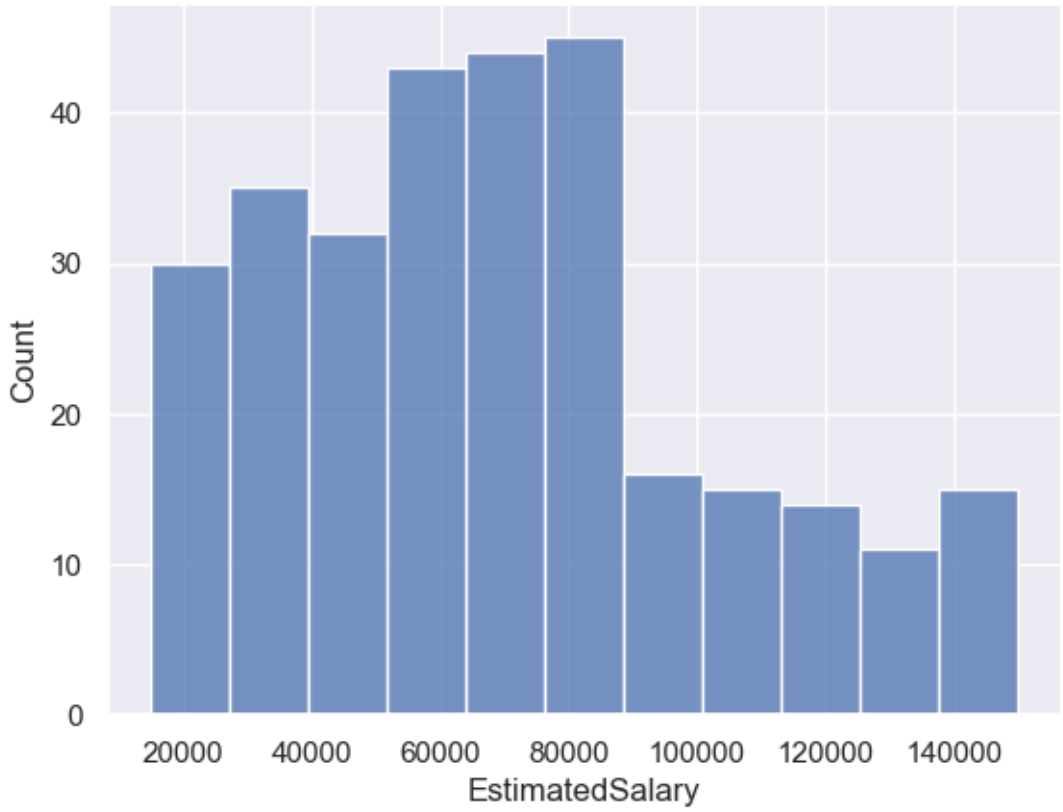
```
LogisticRegression()
```

```
y_pred_test = sklearn_model.predict(X_test)
print('Accuracy of logistic regression on test set : {:.4f}'.format(accuracy_score(y_test, y
```

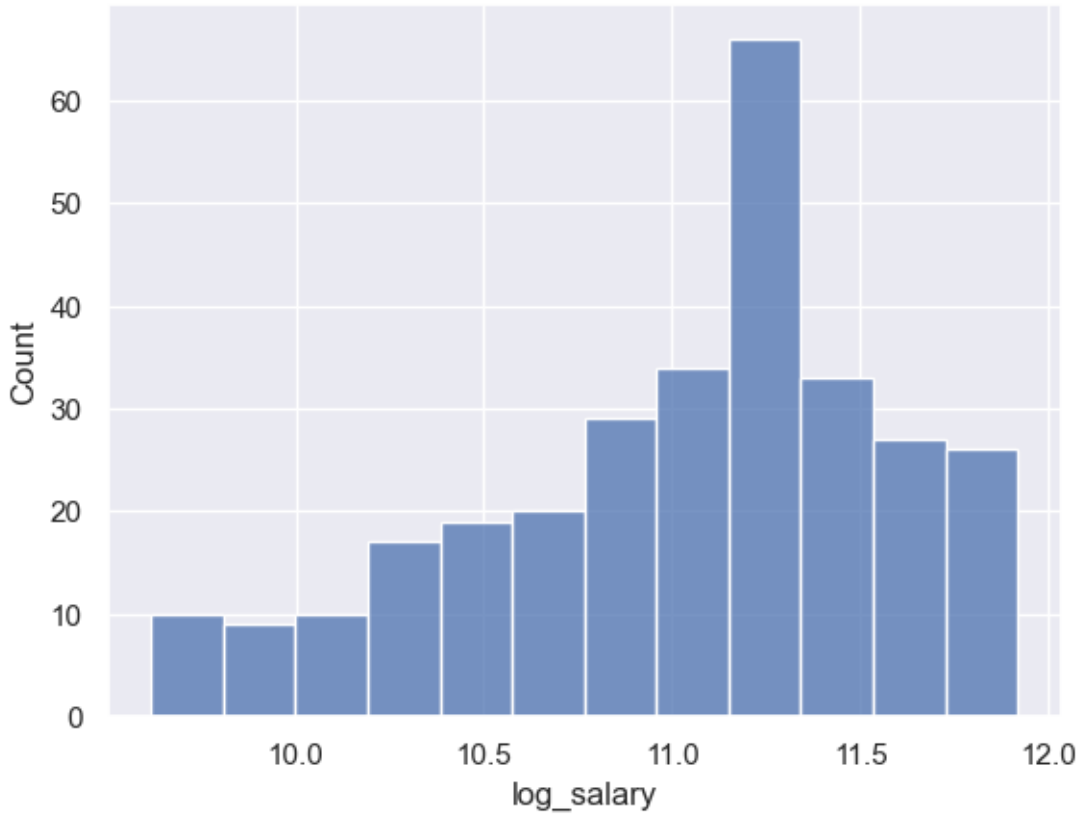
```
Accuracy of logistic regression on test set : 0.8800
```

9.2.2.1 Log transformation of salary

```
sns.histplot(train.EstimatedSalary)
```




```
train["log_salary"] = np.log(train["EstimatedSalary"])
sns.histplot(train.log_salary)
```



The reason for such transformations have nothing to do with their distribution. Instead, the reason has to do with the functional form of the effect. Say we want to know the effect of the number of publications on the probability of getting tenure. It is reasonable to believe that getting an extra publication when one has only 1 publication has more impact compared with getting an extra publication when one has already published 50 articles. The log transformation is one way to capture such a (testable) assumption of diminishing returns.

```
test["log_salary"] = np.log(test["EstimatedSalary"])
```

```
# Separating features and target
y_train = train.Purchased
X_train = train.drop(["Purchased", "Gender", "User ID", "EstimatedSalary"], axis = 1)
```

```
X_train
```

	Age	Male	log_salary
0	36	True	10.404263
1	39	False	11.018629
2	36	True	11.678440
3	39	True	11.711776
4	26	False	11.678440
...	...	...	...
295	48	False	11.472103
296	42	True	11.911702
297	28	True	11.277203
298	51	False	11.805595
299	33	False	10.239960

```
from sklearn.linear_model import LogisticRegression
sklearn_model_log = LogisticRegression()
sklearn_model_log.fit(X_train, y_train)
```

```
LogisticRegression()
```

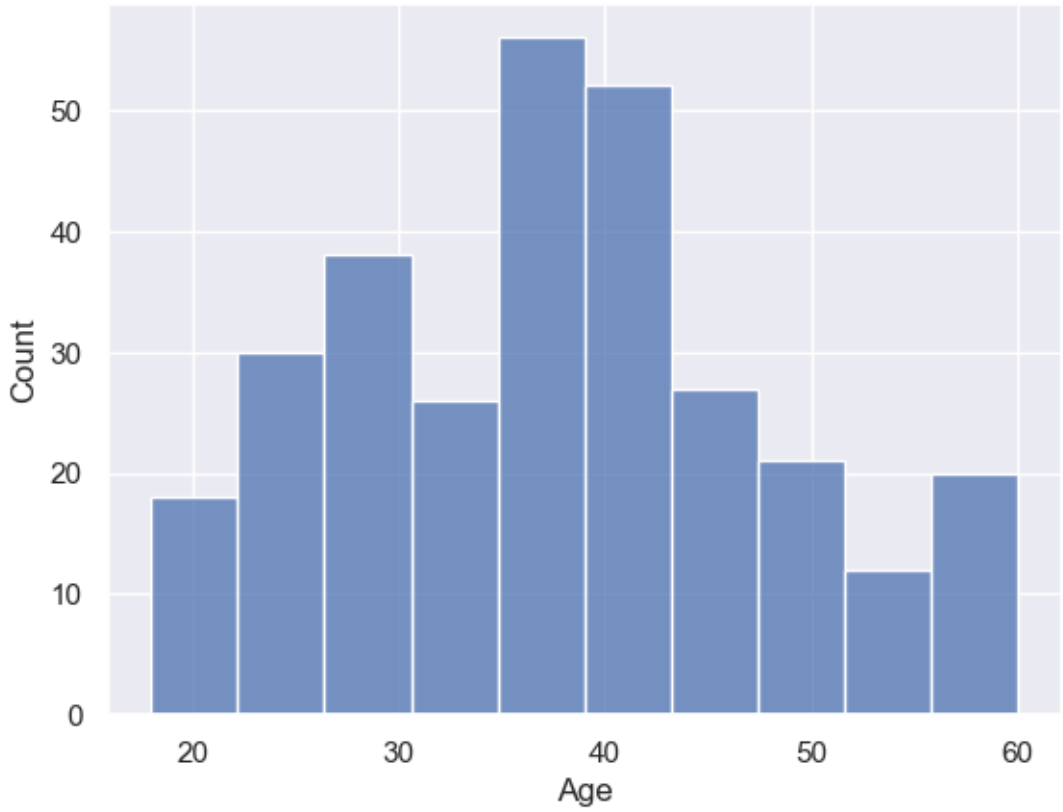
```
# Separating features and target for the test dataset
y_test_log = test.Purchased
X_test_log = test.drop(["Purchased", "Gender", "User ID", "EstimatedSalary"], axis = 1)
```

```
y_log_pred_test = sklearn_model_log.predict(X_test_log)
print('Accuracy of logistic regression after log transformation of salary on test set : {:.4
```

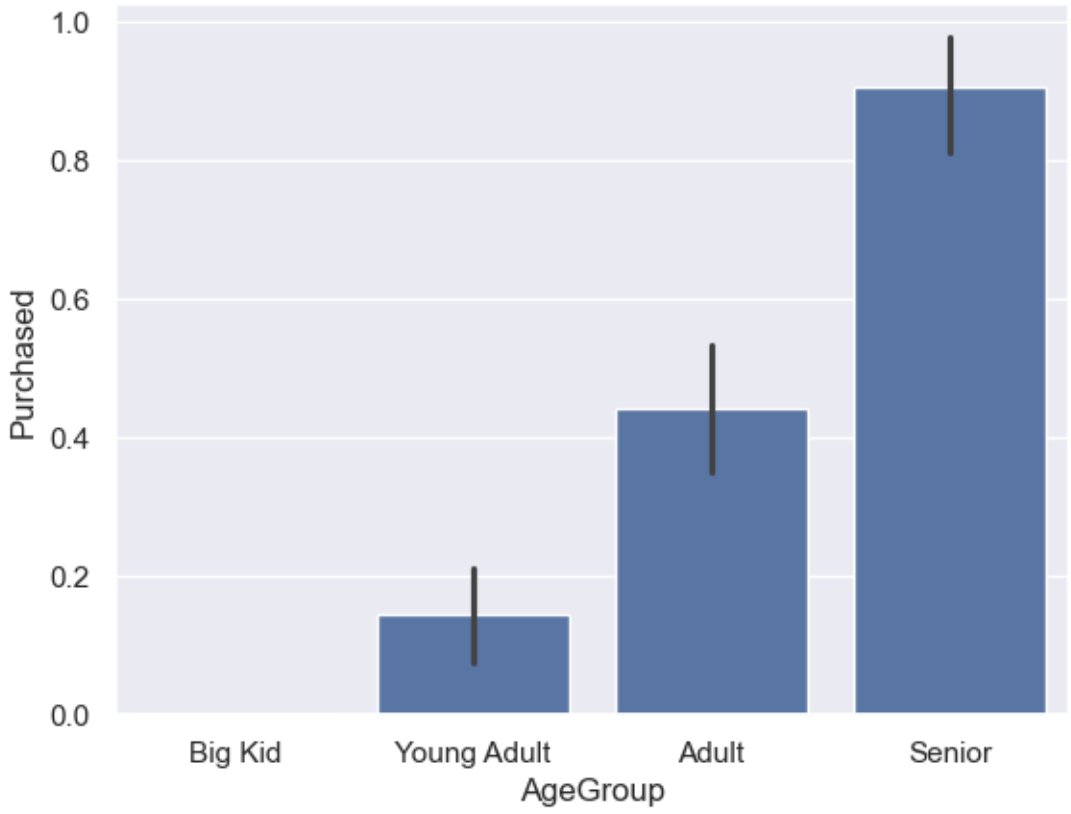
```
Accuracy of logistic regression after log transformation of salary on test set : 0.8300
```

9.2.3 By Binning Continous Variables

```
sns.histplot(data=train.Age)
```




```
bins = [train.Age.min()-1, 25, 35, 48, train.Age.max()]
labels = ['Big Kid', 'Young Adult', 'Adult', 'Senior']
train['AgeGroup'] = pd.cut(train["Age"], bins, labels = labels)

#draw a bar plot of Age vs. survival
sns.barplot(x="AgeGroup", y="Purchased", data=train)
plt.show()
```




```
train
```

	User ID	Gender	Age	EstimatedSalary	Purchased	Male	log_salary	AgeGroup
0	15755018	Male	36	33000	0	True	10.404263	Adult
1	15697020	Female	39	61000	0	False	11.018629	Adult
2	15796351	Male	36	118000	1	True	11.678440	Adult
3	15665760	Male	39	122000	1	True	11.711776	Adult
4	15794661	Female	26	118000	0	False	11.678440	Young Adult
...	...	...	...	...	...	...	...	...
295	15724536	Female	48	96000	1	False	11.472103	Adult
296	15701537	Male	42	149000	1	True	11.911702	Adult
297	15807481	Male	28	79000	0	True	11.277203	Young Adult
298	15603942	Female	51	134000	0	False	11.805595	Senior
299	15690188	Female	33	28000	0	False	10.239960	Young Adult

```
from sklearn import preprocessing
label_encoder = preprocessing.LabelEncoder()
train['AgeGroup']= label_encoder.fit_transform(train['AgeGroup'])
train['AgeGroup'].unique()
```

```
array([0, 3, 1, 2])
```

```
test['AgeGroup'] = pd.cut(test["Age"], bins, labels = labels)
test['AgeGroup']= label_encoder.fit_transform(test['AgeGroup'])
```

```
train
```

	User ID	Gender	Age	EstimatedSalary	Purchased	Male	log_salary	AgeGroup
0	15755018	Male	36	33000	0	True	10.404263	0
1	15697020	Female	39	61000	0	False	11.018629	0
2	15796351	Male	36	118000	1	True	11.678440	0
3	15665760	Male	39	122000	1	True	11.711776	0
4	15794661	Female	26	118000	0	False	11.678440	3
...	...	...	...	...	...	...	...	...
295	15724536	Female	48	96000	1	False	11.472103	0
296	15701537	Male	42	149000	1	True	11.911702	0
297	15807481	Male	28	79000	0	True	11.277203	3
298	15603942	Female	51	134000	0	False	11.805595	2
299	15690188	Female	33	28000	0	False	10.239960	3

```
# Separating features and target on train set
y_train = train.Purchased
X_train = train.drop(["Purchased", "Gender", "User ID", "EstimatedSalary", "Age"], axis = 1)
X_train
```

	Male	log_salary	AgeGroup
0	True	10.404263	0
1	False	11.018629	0
2	True	11.678440	0
3	True	11.711776	0
4	False	11.678440	3
...	...	...	...
295	False	11.472103	0

	Male	log_salary	AgeGroup
296	True	11.911702	0
297	True	11.277203	3
298	False	11.805595	2
299	False	10.239960	3

```
from sklearn.linear_model import LogisticRegression
sklearn_model_bin = LogisticRegression()
sklearn_model_bin.fit(X_train, y_train)
```

```
LogisticRegression()
```

```
# Separating features and target on test set
y_test = test.Purchased
X_test_bin = test.drop(["Purchased", "Gender", "User ID", "EstimatedSalary", "Age"], axis = 1
X_test_bin
```

	Male	log_salary	AgeGroup
0	True	9.903488	3
1	False	10.668955	3
2	False	10.950807	3
3	False	11.918391	3
4	True	11.289782	3
...	...	...	...
95	False	10.571317	2
96	True	10.434116	0
97	True	10.645425	2
98	False	10.985293	0
99	True	10.043249	2

```
y_bin_pred_test = sklearn_model_bin.predict(X_test_bin)
print('Accuracy of logistic regression after age binning on test set : {:.4f}'.format(accurac
```

```
Accuracy of logistic regression after age binning on test set : 0.7100
```

9.3 Reference

- https://www.linkedin.com/pulse/generating-non-linear-decision-boundaries-using-logistic-d-urso/
- https://jermwatt.github.io/machine_learning_refined/notes/10_Nonlinear_intro/10_4_Twoclass.html
- https://www.kaggle.com/code/lzs0047/logistic-regression-non-linear-decision-boundary/edit
- https://www.kaggle.com/code/ashishrane7/logistic-regression-non-linear-decision-boundary/notebook

10 Cross-Validation

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

10.1 Review: Train-Test Split and Its Limitations

10.1.1 Train-Test Split Recap

Throughout this course, we have used the train-test split approach to evaluate models:

- We split the dataset into training and testing sets.
- The training set is used to fit the model.
- The testing set is used to evaluate performance on unseen data.

This approach provides a simple yet effective way to estimate out-of-sample performance. However, it has limitations:

10.1.2 Limitations of Train-Test Split

- **High Variance**: The model's performance depends on which specific observations end up in the training and testing sets. A different split might lead to different results.
- **Data Efficiency**: A portion of the dataset is reserved for testing, meaning the model is not trained on all available data, which can be problematic for small datasets.
- **Instability**: A single split does not always provide a robust estimate of performance, especially when data is imbalanced or noisy.

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

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import roc_auc_score, f1_score, accuracy_score

from sklearn.datasets import make_classification

%matplotlib inline
plt.style.use('ggplot')
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 5, 4
```

```
# Generate synthetic dataset for binary classification
X, y = make_classification(n_samples=1000, n_features=10, random_state=42)
```

Please execute the cell below multiple times. It will be evident that the accuracy score varies with each run due to different observations in both the training and test sets.

```
# use train/test split
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(X, y)

# Define the model
logit_model = LogisticRegression()
logit_model.fit(X_train_class, y_train_class)
y_pred = logit_model.predict(X_test_class)
print(accuracy_score(y_test_class, y_pred))
```

```
0.856
```

10.2 Cross-Validation: Key Concepts

To address the limitations of train-test split, we introduce cross-validation, which provides a more reliable estimate of model performance by using multiple train-test splits.

10.2.1 Steps for K-fold cross-validation

1. Split the dataset into K **equal** partitions (or "folds").
2. Use fold 1 as the **testing set** and the union of the other folds as the **training set**.
3. Calculate **testing accuracy**.
4. Repeat steps 2 and 3 K times, using a **different fold** as the testing set each time.
5. Use the **average testing accuracy** as the estimate of out-of-sample accuracy.

Diagram of **5-fold cross-validation:**

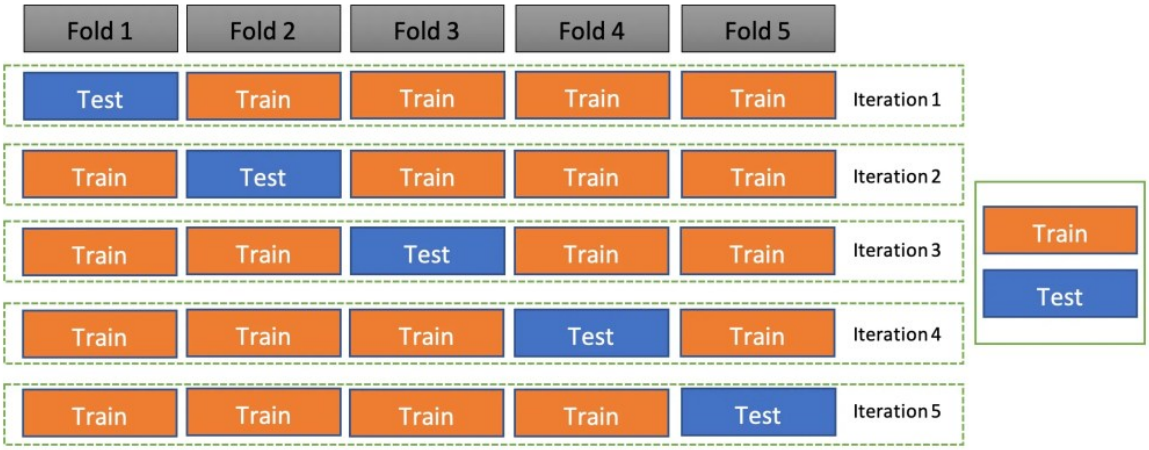


Figure 10.1: 5-fold cross-validation

```
# simulate splitting a dataset of 25 observations into 5 folds
from sklearn.model_selection import KFold
kf = KFold(n_splits=5, shuffle=False).split(range(25))

# print the contents of each training and testing set
print('{} {:^61} {}'.format('Iteration', 'Training set observations', 'Testing set observati
for iteration, data in enumerate(kf, start=1):
    print('{:^9} {} {:^25}'.format(iteration, data[0], str(data[1])))
```

```
Iteration                    Training set observations                   Testing set observati
    1      [ 5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]         [0 1 2 3 4]
    2      [ 0  1  2  3  4 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]         [5 6 7 8 9]
    3      [ 0  1  2  3  4  5  6  7  8  9 15 16 17 18 19 20 21 22 23 24]      [10 11 12 13 14]
    4      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 20 21 22 23 24]      [15 16 17 18 19]
    5      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]      [20 21 22 23 24]
```

Key takeaways

- The dataset consists of **25 observations** (indexed from 0 to 24).
- We use **5-fold cross-validation**, meaning the process runs for **5 iterations** (one per fold).
- In each iteration:

  - The dataset is split into a **training set** and a **testing set**.
  - Each observation is included in either the training set or the testing set, **but never both simultaneously**.

- Over the entire cross-validation process:
  - Each observation appears in the **testing set exactly once**.
  - Each observation is included in the **training set for (K-1) = 4 iterations**.

This ensures that every data point contributes to both model training and evaluation, improving the robustness of performance estimates.

10.3 Cross-Validation in Scikit-Learn

10.3.1 `cross_val_score`

`cross_val_score` is a function in **Scikit-Learn** that simplifies **k-fold cross-validation** for model evaluation. It automates the process of splitting the dataset, training the model, and computing performance metrics across multiple folds.

- By default, it uses **5-fold cross-validation** (`cv=5`).
- For **classification models**, it evaluates performance using **accuracy** as the default scoring metric.
- For **regression models**, it uses **R^2 (coefficient of determination)** by default.
- The function returns an array of scores, one for each fold, providing a more reliable estimate of model performance than a single train-test split.

Using `cross_val_score` ensures a more robust evaluation by reducing variance and making better use of available data.

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(logit_model, X, y)

print(scores)
```

```
[0.87  0.855 0.85  0.83  0.875]
```

Finally, we compute the **mean performance score** across all folds to obtain a robust evaluation.

```
# get the mean score
scores.mean()
```

```
0.8560000000000001
```

By default, `cross_val_score` in **Scikit-Learn** performs **k-fold cross-validation** using the default cross-validator for the given estimator type.

10.3.1.1 Default k-fold cross-validation Settings

`cv=5` unless specified

Model Type	Default Cross-Validator	Shuffling	Stratification
Classification	`StratifiedKFold(n_splits=5)`	No	Yes
Regression	`KFold(n_splits=5)`	No	No

10.3.1.2 How to Modify the Behavior

If you need shuffling or a different cross-validation strategy, specify a custom cross-validator.

10.3.1.2.1 Enable Shuffling

```
# Turn on the shuffle
kf = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(logit_model, X, y, cv=kf, scoring='accuracy')
scores
```

```
array([0.83 , 0.875, 0.87 , 0.85 , 0.865])
```

10.3.1.2.2 Stratified Splitting for Classification

For classification problems, **stratified sampling** is recommended for creating the folds

- Each response class should be represented with equal proportions in each of the K folds

```
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(logit_model, X, y, cv=skf, scoring='accuracy')
scores
```

```
array([0.83 , 0.84 , 0.875, 0.89 , 0.845])
```

10.3.1.2.3 Use Leave-One-Out (LOO) Cross-Validation

If you don't have much data, so any split from the full set to the training and validation set is going to result in really very few observations on which you can train. Leave-on-out cross validation (`LOOCV`) that might work better for cross-validation. Say you have 16 observations. Train on 15 and validate on the other one. Repeat this until you have trained on every set of 15 with the 16th sample left out.

```
from sklearn.model_selection import LeaveOneOut

loo = LeaveOneOut()
scores = cross_val_score(logit_model, X, y, cv=loo, scoring='accuracy')
print(len(scores))
scores[:10]
```

```
1000
```

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

Types of Cross-Validation

Method	Description	Use Case
k-Fold CV	Splits data into `k` folds, each used for training/testing	Most common approach (e.g., `k=5` or `k=10`)
Stratified k-Fold	Like k-Fold, but preserves class proportions	Useful for imbalanced classification
Leave-One-Out (LOO-CV)	Uses **each sample** as a test set once	For very small datasets
Leave-P-Out (LPO-CV)	Leaves out `p` samples for testing in each iteration	Computationally expensive

Method	Description	Use Case
Time Series CV	Ensures training data precedes test data (rolling windows)	Time-series forecasting problems

10.3.1.3 Changing the Scoring Metric in `cross_val_score`

we can specify a different evaluation metric using the `scoring` parameter. Scikit-Learn provides various built-in metrics, including **F1-score**, precision, recall, and more.

You can specify different metrics based on the type of model:

Task Type	Metric Name (for `scoring`)	Description
Classification	`'accuracy'`	Default, ratio of correct predictions
Classification	`'precision'`, `'recall'`	Measure of correctness for positive class
Classification	`'f1'`, `'f1_macro'`	Harmonic mean of precision & recall
Regression	`'neg_mean_squared_error'`	MSE (lower is better)
Regression	`'r2'`	Default, measures variance explained

You can refer to the scikit-learn documentation. If a built-in metric doesn't fit your needs, you can define a custom scoring function

For classification problems, especially imbalanced datasets, **F1-score** is a better metric than accuracy as it considers both **precision and recall**. Let's use it as our metric next

```
# create a KFold object with 5 splits
folds = KFold(n_splits = 5, shuffle = True, random_state = 100)
scores = cross_val_score(logit_model, X, y, cv=folds, scoring= "f1")
scores
```

```
array([0.875     , 0.81632653, 0.88151659, 0.87958115, 0.85869565])
```

10.3.2 `cross_validate` in Scikit-Learn

The `cross_validate` function provides a **more comprehensive evaluation** of a model compared to `cross_val_score`.

Its Key Feature includes:

- Allows **multiple evaluation metrics** to be specified at once.
- Returns a **detailed dictionary** containing:
  - **Training scores** and **test scores** across different folds.
  - **Fit times** (time taken to train the model for each fold).
  - **Score times** (time taken to evaluate the model for each fold).
- Useful for:
  - **Analyzing training/testing time variability** across folds.
  - **Comparing multiple performance metrics simultaneously** to get a more complete picture of model performance.

This function is ideal when you need **deeper insights** into how your model behaves across different folds, beyond just performance scores.

```
from sklearn.model_selection import cross_validate

scores = cross_validate(logit_model, X, y, scoring="accuracy", return_train_score=True)

df_scores = pd.DataFrame(scores)
df_scores
```

	fit_time	score_time	test_score	train_score
0	0.007137	0.001002	0.870	0.86125
1	0.006532	0.000997	0.855	0.86750
2	0.004417	0.000998	0.850	0.87375
3	0.001992	0.001005	0.830	0.87000
4	0.001035	0.000000	0.875	0.85750

Let's use multiple metrics

```
# Define scoring metrics explicitly for multiclass
scoring = ['accuracy', 'recall', 'precision', 'f1', 'roc_auc']

# Perform cross-validation
```

```
scores = cross_validate(logit_model, X, y, scoring=scoring, return_train_score=True)

# Convert to DataFrame for better readability
df_scores = pd.DataFrame(scores)
df_scores
```

	fit_time	score_time	test_accuracy	train_accuracy	test_recall	train_recall	test_precision	trai
0	0.003802	0.000000	0.870	0.86125	0.868687	0.852500	0.868687	0.86
1	0.004556	0.004991	0.855	0.86750	0.810000	0.862155	0.890110	0.87
2	0.003989	0.004989	0.850	0.87375	0.840000	0.867168	0.857143	0.87
3	0.003987	0.004063	0.830	0.87000	0.840000	0.862155	0.823529	0.87
4	0.000000	0.011978	0.875	0.85750	0.890000	0.847118	0.864078	0.86

10.3.3 `cross_val_predict` in sklearn

`cross_val_predict` is a function in Scikit-Learn that performs cross-validation but instead of returning evaluation scores, it returns predicted values for each instance in the dataset as if they were unseen.

10.3.3.1 Process:

1. The dataset is split into **k folds**.
2. The model is trained on **k-1 folds**.
3. Predictions are made on the **remaining fold**.
4. This process repeats **until every instance has been predicted once**, ensuring that each prediction is made on unseen data.

This approach **mimics real-world predictions**, making it useful for evaluating model performance with **classification reports, confusion matrices, and ROC curves**.

```
from sklearn.model_selection import cross_val_predict

# output the predicted probabilities
y_pred_prob = cross_val_predict(logit_model, X, y, cv=5, method='predict_proba')[:,1]
print(y_pred_prob.shape)
print(y_pred_prob[:10])
```

```
(1000,)
[0.04211674 0.97024937 0.96688462 0.01087727 0.65634726 0.02306063
 0.14247751 0.92610262 0.8783967  0.2408313 ]
```

10.3.4 Key Differences Between `cross_val_score`, `cross_validate`, and `cross_val_predict`

Function	Returns	Requires `scoring`?	Supports Multiple Metrics?	Can Return Train Scores?	Purpose
`cross_val_score`	Array of scores	No (defaults to accuracy for classification)	No	No	Evaluates model performance using cross-validation
`cross_validate`	Dictionary	Yes (must specify)	Yes (`scoring={'accuracy', 'precision'}`)	Yes (`return_train_score=True`)	Provides detailed evaluation metrics, including training times
`cross_val_predict`	Array of predictions	No (uses `predict` or `predict_proba`)	No	No	Generates out-of-sample predictions for each instance

10.3.5 Advantages and Disadvantages of Cross-Validation

10.3.5.1 Advantages:

- **Reduces Variance**: By averaging results over multiple folds, cross-validation provides a more stable estimate of model performance.
- **Better Use of Data**: Every observation gets a chance to be in both the training and testing sets, improving data efficiency.
- **More Reliable Performance Metrics**: The results are less dependent on a single random split, making the evaluation more robust.

10.3.5.2 Disadvantages Compared to Train-Test Split:

- **Higher Computational Cost**: Instead of training the model once (as in a train-test split), cross-validation requires training the model **k times**, making it computationally expensive for large datasets.
- **Potential Overfitting to Small Data**: If `k` is too large (e.g., Leave-One-Out Cross-Validation), it can lead to high variance and make the model too sensitive to small changes in data.
- **Choosing `k`**: Typically, **`k=5` or `k=10`** is recommended as a balance between bias and variance.

Cross-validation is **preferable** for small-to-moderate datasets where stable performance estimates are important, while a simple **train-test split** is often sufficient for large datasets when computational efficiency is a priority.

10.4 Cross-Validation for Hyperparameter Tuning

Cross-validation is a powerful technique that can be used for:

- **Model Performance Evaluation**: Provides a more reliable estimate of how well a model generalizes to unseen data.
- **Hyperparameter Tuning**: Helps find the optimal model parameters by evaluating different configurations.

- **Model Selection**: Compares multiple models to choose the one that performs best across different folds.

- **Feature Selection**: Assesses the impact of different feature subsets on model performance.

In this notebook, we focus exclusively on **Hyperparameter Tuning with Cross-Validation**.
We demonstrate how cross-validation can be used to systematically search for the best hyperparameters, ensuring **better generalization** and **optimized model performance**.

10.4.1 Finding the optimal Degree in Polynomial Regression

10.4.1.1 Background: Polynomial Regression

You already know simple linear regression:

$y = \beta_0 + \beta_1 x_1$

In polynomial regression of degree n, we fit a curve of the form:

$y = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_1^3 ... + \beta_n x_1^n$

In the experiment below, we fit polynomial models of varying degrees on simulated data to analyze their performance.
We build a **linear regression model** and use **cross-validation** to tune the polynomial degree (`p`).

By selecting the optimal degree, we aim to **balance the trade-off between underfitting and overfitting**, ensuring the model generalizes well to unseen data.

Specifically, we Will Cover:

- **Using `cross_val_score`** for hyperparameter tuning to evaluate model performance across folds.

- **Using `cross_validate`** to obtain detailed metrics, including training scores and fit times.

- **Applying `GridSearchCV`** to systematically search for the optimal polynomial degree.

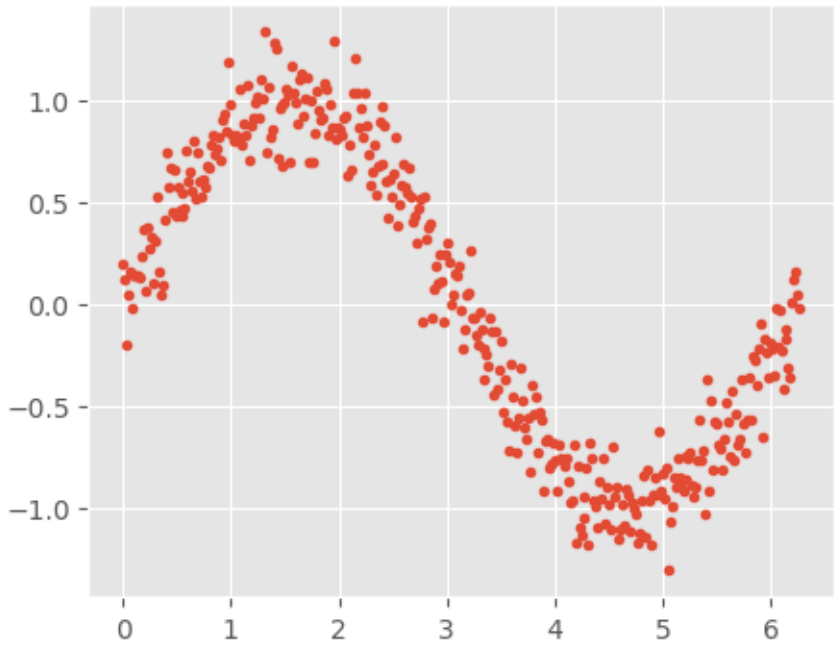
10.4.1.2 Finding the Optimal Degree with cross_val_score

10.4.1.2.1 Step 1: Let's begin by importing the required libraries.

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

from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, PolynomialFeatures, OneHotEncoder
```

10.4.1.2.2 Step 2: Let's generate input data

```
# Simulate input data

#Define input array with angles from 60deg to 300deg converted to radians
x = np.array([i*np.pi/180 for i in range(360)])
np.random.seed(10)  #Setting seed for reproducibility
y = np.sin(x) + np.random.normal(0,0.15,len(x))
data = pd.DataFrame(np.column_stack([x,y]),columns=['x','y'])
plt.plot(data['x'],data['y'],'.');
```



10.4.1.2.3 Step 3: Train-Test Split

```
# Split the data
X = data['x'].values.reshape(-1, 1)
y = data['y'].values

# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

10.4.1.2.4 Step 4: Using `cross_val_score` to tune the p in polynomialFeatures

In sklearn, polynomial features can be generated using the `PolynomialFeatures` class. Also, to perform `LinearRegression` and `PolynomialFeatures` in tandem, we will use the module `sklearn_pipeline` - it basically creates the features and feeds the output to the model (in that sequence).

```
def cross_validation_score(X_train, y_train, max_degree=10, scoring='r2'):
    """
    Perform cross-validation for polynomial regression models with different degrees.

    Parameters:
        X_train (array-like): Training feature data.
        y_train (array-like): Training target data.
        max_degree (int): Maximum polynomial degree to evaluate.
        scoring (str): Scoring metric ('r2' for R², 'rmse' for root mean squared error).

    Returns:
        degrees (list): List of polynomial degrees evaluated.
        scores_df (DataFrame): DataFrame of cross-validation scores across different degrees
    """
    degrees = range(1, max_degree + 1)
    cv_scores = []

    for degree in degrees:
        # Create polynomial regression model
        model = make_pipeline(
            PolynomialFeatures(degree),
            LinearRegression()
        )

        # Compute cross-validation scores
        if scoring == 'rmse':
```

```
            raw_scores = cross_val_score(model, X_train, y_train, cv=5, scoring='neg_mean_squ
            cv_score = np.sqrt(-raw_scores)  # Convert negative MSE to RMSE
            score_label = "RMSE (lower is better)"
        else:  # Default to R²
            cv_score = cross_val_score(model, X_train, y_train, cv=5, scoring='r2')
            score_label = "R² (higher is better)"

        cv_scores.append(cv_score)

    # Convert scores to a DataFrame
    scores_df = pd.DataFrame(np.array(cv_scores), index=degrees, columns=[f'Fold {i+1}' for i

    print(f"Cross-validation scores ({score_label}):")
    return degrees, scores_df
```

```
# Example Usage:
scoring_metric = 'rmse'  # Change to 'r2' if needed
degrees, scores = cross_validation_score(X_train, y_train, scoring="rmse")

# Print the formatted score matrix
print(scores)
```

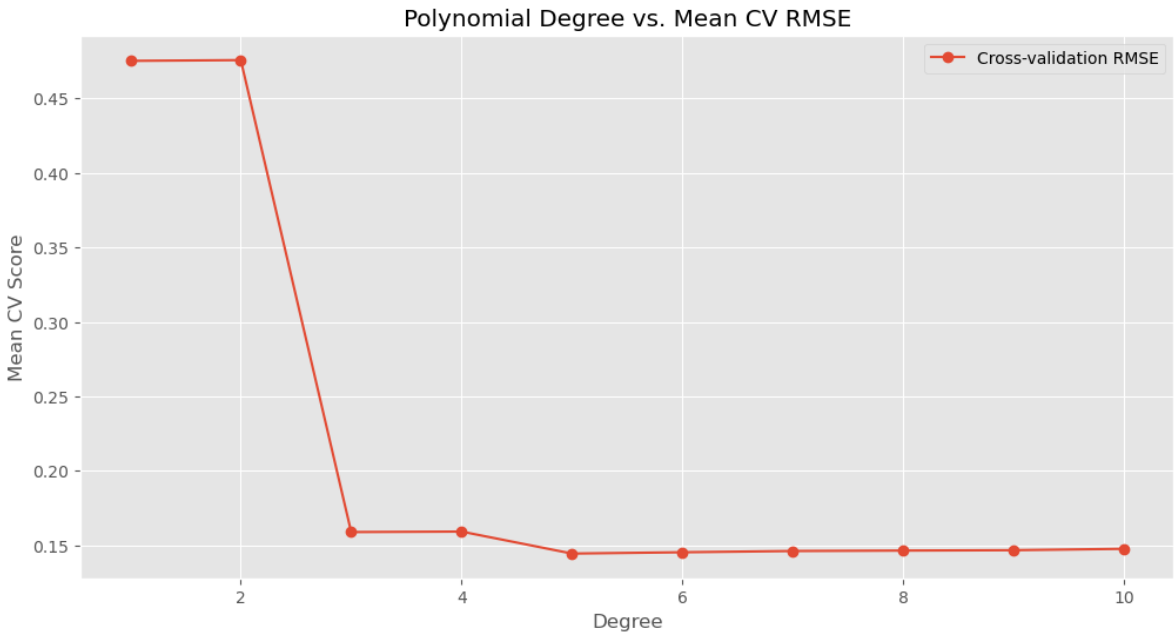
```
Cross-validation scores (RMSE (lower is better)):
      Fold 1    Fold 2    Fold 3    Fold 4    Fold 5
1   0.500794  0.456810  0.472232  0.484231  0.461889
2   0.500255  0.455897  0.475378  0.485450  0.461456
3   0.164968  0.158555  0.156423  0.152829  0.161931
4   0.165521  0.158619  0.156413  0.153359  0.162379
5   0.145887  0.138576  0.143649  0.141445  0.152968
6   0.145995  0.139056  0.143390  0.144567  0.153870
7   0.151113  0.138186  0.144122  0.144211  0.153438
8   0.150406  0.137350  0.143297  0.146944  0.154557
9   0.151095  0.138084  0.143219  0.146828  0.154599
10  0.152122  0.139420  0.145089  0.147192  0.154581
```

```
# take mean of each row and add it as the last column of the scores dataframe
scores['mean'] = scores.mean(axis=1)
print(scores)
```

```
      Fold 1    Fold 2    Fold 3    Fold 4    Fold 5      mean
1   0.500794  0.456810  0.472232  0.484231  0.461889  0.475191
```

```
2   0.500255  0.455897  0.475378  0.485450  0.461456  0.475687
3   0.164968  0.158555  0.156423  0.152829  0.161931  0.158941
4   0.165521  0.158619  0.156413  0.153359  0.162379  0.159258
5   0.145887  0.138576  0.143649  0.141445  0.152968  0.144505
6   0.145995  0.139056  0.143390  0.144567  0.153870  0.145376
7   0.151113  0.138186  0.144122  0.144211  0.153438  0.146214
8   0.150406  0.137350  0.143297  0.146944  0.154557  0.146511
9   0.151095  0.138084  0.143219  0.146828  0.154599  0.146765
10  0.152122  0.139420  0.145089  0.147192  0.154581  0.147681
```

```
# plot the mean scores for each degree
plt.figure(figsize=(12, 6))
plt.plot(degrees, scores['mean'], 'o-', label='Cross-validation RMSE')
plt.xlabel('Degree')
plt.ylabel('Mean CV Score')
plt.title('Polynomial Degree vs. Mean CV RMSE')
plt.grid(True)
plt.legend();
```




```
# Find optimal degree based on the mean cross-validation scores
optimal_degree = scores['mean'].idxmin()
print(f'Optimal polynomial degree: {optimal_degree}')
```

```
Optimal polynomial degree: 5
```

```
# use r2 as performance metric, note that r2 is default metric in the function defination
degrees, scores = cross_validation_score(X_train, y_train)

# Print the formatted score matrix
print(scores)
```

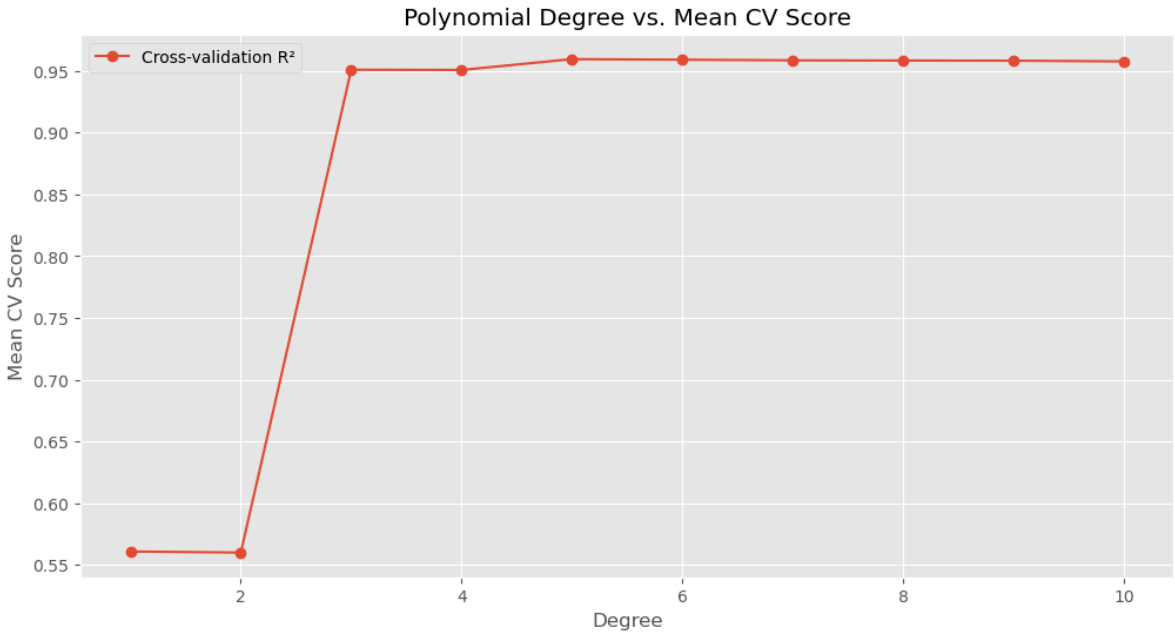
```
Cross-validation scores (R² (higher is better)):
      Fold 1    Fold 2    Fold 3    Fold 4    Fold 5
1   0.474906  0.587650  0.567349  0.584471  0.590389
2   0.476036  0.589297  0.561564  0.582378  0.591156
3   0.943020  0.950323  0.952529  0.958609  0.949655
4   0.942638  0.950283  0.952535  0.958321  0.949376
5   0.955439  0.962054  0.959965  0.964546  0.955074
6   0.955373  0.961790  0.960110  0.962963  0.954543
7   0.952190  0.962267  0.959701  0.963145  0.954797
8   0.952636  0.962722  0.960162  0.961735  0.954136
9   0.952201  0.962322  0.960205  0.961795  0.954111
10  0.951549  0.961590  0.959159  0.961606  0.954121
```

```
# take mean of each row and add it as the last column of the scores dataframe
scores['mean'] = scores.mean(axis=1)
print(scores)
```

```
      Fold 1    Fold 2    Fold 3    Fold 4    Fold 5      mean
1   0.474906  0.587650  0.567349  0.584471  0.590389  0.560953
2   0.476036  0.589297  0.561564  0.582378  0.591156  0.560086
3   0.943020  0.950323  0.952529  0.958609  0.949655  0.950827
4   0.942638  0.950283  0.952535  0.958321  0.949376  0.950631
5   0.955439  0.962054  0.959965  0.964546  0.955074  0.959416
6   0.955373  0.961790  0.960110  0.962963  0.954543  0.958956
7   0.952190  0.962267  0.959701  0.963145  0.954797  0.958420
8   0.952636  0.962722  0.960162  0.961735  0.954136  0.958278
9   0.952201  0.962322  0.960205  0.961795  0.954111  0.958127
10  0.951549  0.961590  0.959159  0.961606  0.954121  0.957605
```

```
# plot the mean scores for each degree
plt.figure(figsize=(12, 6))
plt.plot(degrees, scores['mean'], 'o-', label='Cross-validation R²')
plt.xlabel('Degree')
```

```
plt.ylabel('Mean CV Score')
plt.title('Polynomial Degree vs. Mean CV Score')
plt.grid(True)
plt.legend();
```




```
# Find optimal degree based on the mean cross-validation scores
optimal_degree = scores['mean'].idxmax()
print(f'Optimal polynomial degree: {optimal_degree}')
```

```
Optimal polynomial degree: 5
```

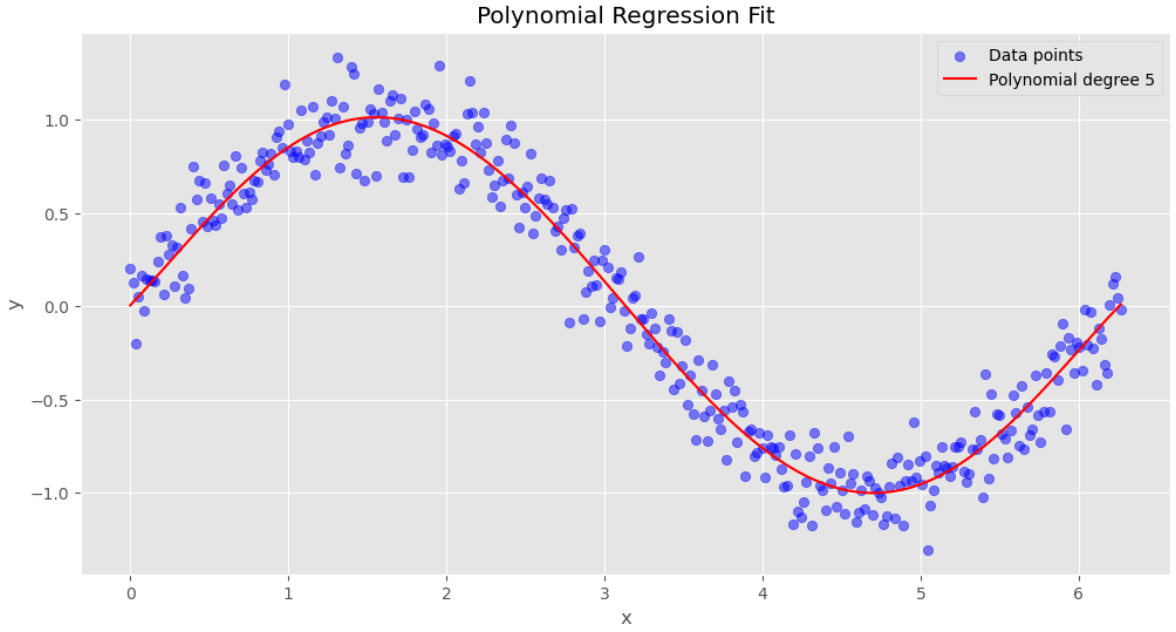
10.4.1.2.5 Step 5: Fitting the Final Model with the Optimal Polynomial Degree

After determining the **optimal polynomial degree (p=5)** using cross-validation, we now **refit the model** on the entire training dataset.
This ensures that the model fully utilizes all available training data for the best possible fit.

Steps: 1. **Build the final model** using a polynomial transformation with the optimal degree (p=5). 2. **Train the model** on the full training dataset. 3. **Generate predictions** and visualize how well the polynomial regression fits the data.

The plot below shows the **final polynomial regression fit**, highlighting how the model captures patterns in the dataset.

```
# Fit final model with optimal degree
final_model = make_pipeline(
    PolynomialFeatures(optimal_degree),
    LinearRegression()
)
final_model.fit(X_train, y_train)

# Plot final model predictions
plt.figure(figsize=(12, 6))
X_sorted = np.sort(X)
y_pred = final_model.predict(X_sorted)

plt.scatter(X, y, color='blue', alpha=0.5, label='Data points')
plt.plot(X_sorted, y_pred, color='red', label=f'Polynomial degree {optimal_degree}')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Polynomial Regression Fit')
plt.legend()
plt.grid(True)
plt.show()
```



10.4.1.2.6 Step 6: Output the final model performance

```
# Print final model performance metrics
print("\nFinal Model Performance:")
print(f"Training R²: {final_model.score(X_train, y_train):.4f}")
print(f"Test R²: {final_model.score(X_test, y_test):.4f}")
```

```
Final Model Performance:
Training R²: 0.9613
Test R²: 0.9521
```

10.4.1.3 Hyperparameter Tuning with cross_validate

The `cross_validate` function differs from cross_val_score in two ways:

- It allows specifying multiple metrics for evaluation.
- It returns a dict containing fit-times, score-times (and optionally training scores, fitted estimators, train-test split indices) in addition to the test score.

```
from sklearn.model_selection import cross_validate

def cross_validation_full_results(X_train, y_train, max_degree=10):
    """
    Perform cross-validation for polynomial regression models with different degrees.
    Captures all cross-validation results including R², RMSE, fit time, score time, etc.

    Parameters:
        X_train (array-like): Training feature data.
        y_train (array-like): Training target data.
        max_degree (int): Maximum polynomial degree to evaluate.

    Returns:
        full_results_df (DataFrame): DataFrame containing all cross-validation metrics for ea
    """
    degrees = []
    results_dict = {
        "Fold": [],
        "Fit Time": [],
        "Score Time": [],
        "R²": [],
        "RMSE": []
    }
```

```
    for degree in range(1, max_degree + 1):
        # Create polynomial regression model
        model = make_pipeline(
            PolynomialFeatures(degree),
            LinearRegression()
        )

        # Perform cross-validation for both R² and RMSE, capturing additional metrics
        cv_results = cross_validate(model, X_train, y_train, cv=5,
                                    scoring=['r2', 'neg_root_mean_squared_error'], return_tra

        # Store each fold's results separately
        for fold in range(5):  # 5 folds
            degrees.append(degree)
            results_dict["Fold"].append(fold + 1)
            results_dict["Fit Time"].append(cv_results["fit_time"][fold])
            results_dict["Score Time"].append(cv_results["score_time"][fold])
            results_dict["R²"].append(cv_results["test_r2"][fold])
            results_dict["RMSE"].append(-cv_results["test_neg_root_mean_squared_error"][fold]

    # Convert to DataFrame
    full_results_df = pd.DataFrame(results_dict)
    full_results_df.insert(0, "Degree", degrees)  # Add Degree column at the front

    print("Complete Cross-Validation Results (Higher R² is better, Lower RMSE is better, Fit
    return full_results_df

scores_df = cross_validation_full_results(X_train, y_train, max_degree=10)

scores_df
```

```
Complete Cross-Validation Results (Higher R² is better, Lower RMSE is better, Fit Time & Scor
```

	Degree	Fold	Fit Time	Score Time	R^2	RMSE
0	1	1	0.001993	0.002523	0.474906	0.500794
1	1	2	0.001994	0.002033	0.587650	0.456810
2	1	3	0.000000	0.000000	0.567349	0.472232
3	1	4	0.000996	0.000996	0.584471	0.484231
4	1	5	0.000997	0.001993	0.590389	0.461889
5	2	1	0.001482	0.000000	0.476036	0.500255

	Degree	Fold	Fit Time	Score Time	R^2	RMSE
6	2	2	0.006236	0.002004	0.589297	0.455897
7	2	3	0.000996	0.000997	0.561564	0.475378
8	2	4	0.000997	0.000997	0.582378	0.485450
9	2	5	0.000997	0.000997	0.591156	0.461456
10	3	1	0.000996	0.000997	0.943020	0.164968
11	3	2	0.000997	0.001250	0.950323	0.158555
12	3	3	0.001089	0.000000	0.952529	0.156423
13	3	4	0.000000	0.000000	0.958609	0.152829
14	3	5	0.000000	0.000000	0.949655	0.161931
15	4	1	0.000000	0.000000	0.942638	0.165521
16	4	2	0.000000	0.000000	0.950283	0.158619
17	4	3	0.000996	0.000996	0.952535	0.156413
18	4	4	0.000997	0.000996	0.958321	0.153359
19	4	5	0.000997	0.000997	0.949376	0.162379
20	5	1	0.000996	0.000992	0.955439	0.145887
21	5	2	0.000997	0.001001	0.962054	0.138576
22	5	3	0.000997	0.000993	0.959965	0.143649
23	5	4	0.000996	0.000997	0.964546	0.141445
24	5	5	0.000000	0.000000	0.955074	0.152968
25	6	1	0.000000	0.000000	0.955373	0.145995
26	6	2	0.000000	0.000000	0.961790	0.139056
27	6	3	0.000000	0.000000	0.960110	0.143390
28	6	4	0.000000	0.000996	0.962963	0.144567
29	6	5	0.000000	0.000997	0.954543	0.153870
30	7	1	0.000998	0.001304	0.952190	0.151113
31	7	2	0.000997	0.001003	0.962267	0.138186
32	7	3	0.000000	0.000997	0.959701	0.144122
33	7	4	0.000000	0.000993	0.963145	0.144211
34	7	5	0.000997	0.000997	0.954797	0.153438
35	8	1	0.000997	0.000842	0.952636	0.150406
36	8	2	0.000000	0.000000	0.962722	0.137350
37	8	3	0.000000	0.000000	0.960162	0.143297
38	8	4	0.010667	0.000995	0.961735	0.146944
39	8	5	0.000000	0.000000	0.954136	0.154557
40	9	1	0.000000	0.000000	0.952201	0.151095
41	9	2	0.000000	0.000000	0.962322	0.138084
42	9	3	0.000000	0.000000	0.960205	0.143219
43	9	4	0.000996	0.001993	0.961795	0.146828
44	9	5	0.000170	0.000000	0.954111	0.154599
45	10	1	0.000000	0.000000	0.951549	0.152122
46	10	2	0.000000	0.000000	0.961590	0.139420

	Degree	Fold	Fit Time	Score Time	R^2	RMSE
47	10	3	0.000000	0.012748	0.959159	0.145089
48	10	4	0.000996	0.001614	0.961606	0.147192
49	10	5	0.000000	0.000000	0.954121	0.154581

```
mean_results_df = scores_df.groupby("Degree")[["R²", "RMSE"]].mean().reset_index()
mean_results_df
```

	Degree	R^2	RMSE
0	1	0.560953	0.475191
1	2	0.560086	0.475687
2	3	0.950827	0.158941
3	4	0.950631	0.159258
4	5	0.959416	0.144505
5	6	0.958956	0.145376
6	7	0.958420	0.146214
7	8	0.958278	0.146511
8	9	0.958127	0.146765
9	10	0.957605	0.147681

10.4.1.4 Grid Search Cross-Validation for Hyperparameter Tuning

A common use of **cross-validation** is tuning the hyperparameters of a model. One of the most widely used techniques for this is **grid search cross-validation**.

How Grid Search Cross-Validation Works

1. **Define a Grid of Hyperparameters:**
   We specify a set of hyperparameters and the possible values we want to evaluate for each.
2. **Evaluate All Combinations:**
   Every possible combination of hyperparameter values in the grid is systematically tested.
3. **Cross-Validation for Model Evaluation:**
   For each combination, the model is trained and evaluated using cross-validation to assess performance.
4. **Select the Best Hyperparameter Setting:**
   The combination that yields the best validation performance is chosen as the optimal set of hyperparameters.

Grid search ensures that we explore multiple hyperparameter settings in a structured way, improving the model's performance without manually adjusting parameters.

```
from sklearn.model_selection import GridSearchCV

# Define the pipeline
model = make_pipeline(PolynomialFeatures(), LinearRegression())

# Define the hyperparameter grid
param_grid = {'polynomialfeatures__degree': np.arange(1, 11)}

# define a KFold cross-validation with shuffling
cv = KFold(n_splits=5, shuffle=True, random_state=42)


# Perform Grid Search with Cross-Validation
grid_search = GridSearchCV(model, param_grid, cv=cv, scoring= 'r2', return_train_score=True,

# Fit the model
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits

GridSearchCV(cv=KFold(n_splits=5, random_state=42, shuffle=True),
             estimator=Pipeline(steps=[('polynomialfeatures',
                                        PolynomialFeatures()),
                                       ('linearregression',
                                        LinearRegression())]),
             n_jobs=-1,
             param_grid={'polynomialfeatures__degree': array([ 1,  2,  3,  4,  5,  6,  7,  8
             return_train_score=True, scoring='r2', verbose=1)
```

```
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results
```

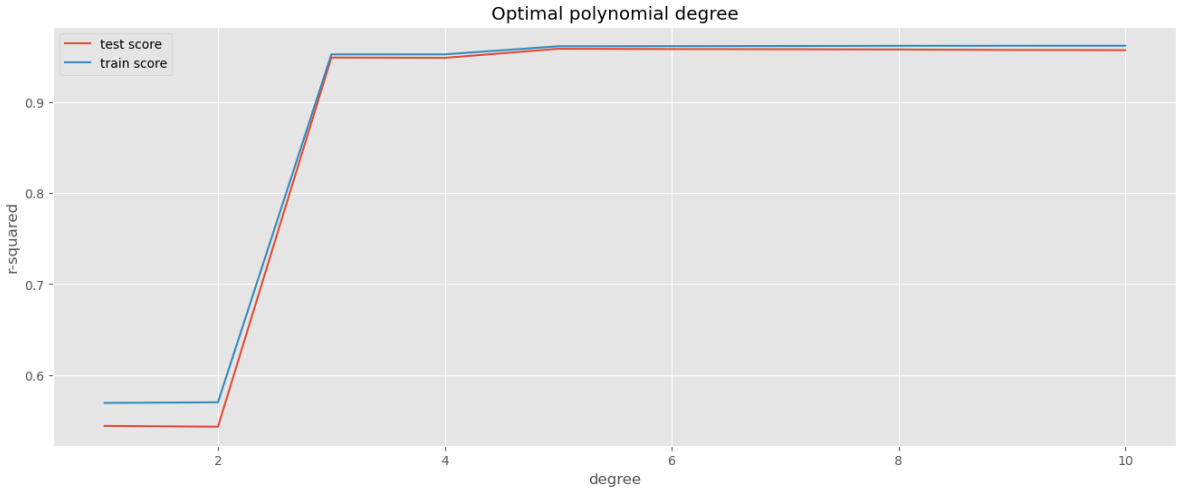
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_polynomialfeatures___degr
0	0.014676	0.001534	0.002452	0.000697	1
1	0.014464	0.001952	0.002246	0.000386	2
2	0.010744	0.001704	0.001196	0.000399	3
3	0.003352	0.001858	0.001900	0.000197	4
4	0.003586	0.002212	0.001844	0.000487	5
5	0.002876	0.001588	0.001994	0.000631	6
6	0.003966	0.003833	0.001792	0.000374	7

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_polynomialfeatures___degr
7	0.002766	0.000773	0.001784	0.000394	8
8	0.002394	0.000487	0.001599	0.000487	9
9	0.002936	0.000066	0.001993	0.000036	10

```
pd.set_option('display.float_format', '{:.6f}'.format)
cv_results[["mean_test_score", "mean_train_score"]]
```

	mean_test_score	mean_train_score
0	0.543991	0.569256
1	0.543109	0.569957
2	0.948949	0.952430
3	0.948634	0.952467
4	0.958620	0.961378
5	0.958278	0.961477
6	0.958136	0.961659
7	0.957822	0.961856
8	0.957306	0.961933
9	0.957090	0.961955

```
# plotting cv results
plt.figure(figsize=(16,6))

plt.plot(cv_results["param_polynomialfeatures__degree"], cv_results["mean_test_score"])
plt.plot(cv_results["param_polynomialfeatures__degree"], cv_results["mean_train_score"])
plt.xlabel('degree')
plt.ylabel('r-squared')
plt.title("Optimal polynomial degree")
plt.legend(['test score', 'train score'], loc='upper left');
```




```
# print out the best hyperparameters and the best score
print(grid_search.best_params_)
print(grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
{'polynomialfeatures__degree': 5}
0.9586200484884486
Pipeline(steps=[('polynomialfeatures', PolynomialFeatures(degree=5)),
                ('linearregression', LinearRegression())])
```

Benefits of `GridSearchCV`

- No need for manually writing nested `for` loops for hyperparameter tuning.
- Allows parallel processing using **multiple CPU cores** (`n_jobs=-1`), speeding up the search.
- `GridSearchCV` guarantees finding the best combination.

However, **GridSearchCV** performs an **exhaustive search** over all possible hyperparameter combinations, ensuring that the best parameters are found. This can be slow, especially when dealing with multiple hyperparameters.

10.4.1.5 GridSearchCV vs. Faster Alternatives for Hyperparameter Tuning

Two Ways to Speed Up Hyperparameter Tuning

1. **RandomizedSearchCV**
    - Instead of checking every combination, it **randomly samples** a set number of hyperparameter values, significantly reducing computation time.
    - Suitable when a rough estimate of the best hyperparameters is sufficient.
2. **ShuffleSplit**
    - Unlike `KFold`, which ensures each sample is used exactly once as a test set, **ShuffleSplit randomly selects train-test splits** in each iteration.
    - Reduces redundant computations, making the process faster while maintaining good model performance.

Both approaches can be combined with `n_jobs=-1` to leverage parallel processing for even faster results.

10.4.2 Tuning the classification threshold

By default, classifiers use `0.5` as the threshold for classification. However, adjusting this threshold can improve `precision`, `recall`, or `F1-score` depending on the application. Let's use cross_val_predict to tune the Classification Threshold

10.4.2.1 Using `TunedThresholdClassifierCV` from sklearn (>= Version 1.5)

```
from sklearn.model_selection import TunedThresholdClassifierCV

# Define cross-validation strategy
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)


# Define TunedThresholdClassifierCV
tuned_clf = TunedThresholdClassifierCV(
    estimator=logit_model,
    scoring="f1",  # Optimize for F1-score
    cv=cv
)

# Fit the model
tuned_clf.fit(X_train_class, y_train_class)
```

```
# Print the best threshold and the corresponding score
print(f"Best threshold: {tuned_clf.best_threshold_:.2f}")
print(f"Best F1-score: {tuned_clf.best_score_:.4f}")
```

```
Best threshold: 0.39
Best F1-score: 0.8757
```

10.4.2.2 Using `cross_val_predict`

```
# use the knn to tune the threshold for classification, the threshold is among np.arange(0.1
thresholds = np.arange(0.1, 0.9, 0.1)
scores = []
for threshold in thresholds:
    y_pred_class_prob = cross_val_predict(logit_model, X_train_class, y_train_class, cv=5, me
    y_pred_class = (y_pred_class_prob > threshold).astype(int)
    scores.append(accuracy_score(y_train_class, y_pred_class))
scores = pd.Series(scores, index=thresholds)

df_scores = pd.DataFrame({'threshold': thresholds, 'accuracy': scores}, )
print(df_scores.to_string(index=False))
```

```
 threshold  accuracy
  0.100000  0.716000
  0.200000  0.832000
  0.300000  0.864000
  0.400000  0.868000
  0.500000  0.865333
  0.600000  0.861333
  0.700000  0.841333
  0.800000  0.810667
```

```
# print out the best threshold  and the cv score for the best threshold
best_threshold = scores.idxmax()
print(best_threshold)
print(scores.loc[best_threshold])
```

```
0.4
0.868
```

11 Regularization

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

```
import numpy as np
import pandas as pd
import random
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

%matplotlib inline
plt.style.use('ggplot')
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 5, 4
```

11.1 Why do we need regularization?

11.1.1 The Challenge of Overfitting and Underfitting

When building machine learning models, we aim to find patterns in data that generalize well to unseen samples. However, models can suffer from two key issues:

- **Underfitting**: The model is too simple to capture the underlying pattern in the data.
- **Overfitting**: The model is too complex and captures noise rather than generalizable patterns.

Regularization is a technique used to address overfitting by penalizing overly complex models.

11.1.2 Understanding the Bias-Variance Tradeoff

A well-performing model balances two competing sources of error:

- **Bias**: Error due to overly simplistic assumptions (e.g., underfitting).
- **Variance**: Error due to excessive sensitivity to training data (e.g., overfitting).

A **high-bias** model (e.g., a simple linear regression) may not capture the underlying trend, while a **high-variance** model (e.g., a deep neural network with many parameters) may memorize noise instead of learning meaningful patterns.

Regularization helps **reduce variance** while maintaining an appropriate level of model complexity.

11.1.3 Visualizing Overfitting vs. Underfitting

To better understand this concept, consider three different models:

1. **Underfitting (High Bias)**: The model is too simple and fails to capture important trends.
2. **Good Fit (Balanced Bias & Variance)**: The model generalizes well to unseen data.
3. **Overfitting (High Variance)**: The model is too complex and captures noise, leading to poor generalization.

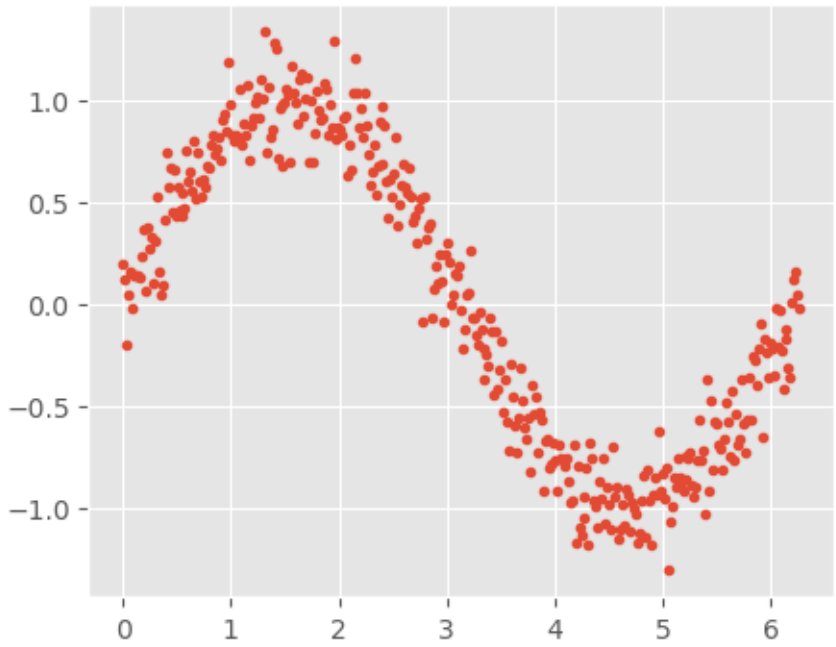
$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

Regularization helps control **variance** by penalizing large coefficients, leading to a model that generalizes better.

11.2 Simulating Data for an Overfitting Linear Model

11.2.1 Generating the data

```
#Define input array with angles from 60deg to 300deg converted to radians
x = np.array([i*np.pi/180 for i in range(360)])
np.random.seed(10)  #Setting seed for reproducibility
y = np.sin(x) + np.random.normal(0,0.15,len(x))
data = pd.DataFrame(np.column_stack([x,y]),columns=['x','y'])
plt.plot(data['x'],data['y'],'.');
```




```
# check how the data looks like
data.head()
```

	x	y
0	0.000000	0.199738
1	0.017453	0.124744
2	0.034907	-0.196911
3	0.052360	0.051078
4	0.069813	0.162957

Polynomial features allow linear regression to model non-linear relationships. Higher-degree terms capture more complex patterns in the data. Let's manually expands features, similar to `PolynomialFeatures` in `sklearn.preprocessing`. Using polynomial regression, we can evaluate different polynomial degrees and analyze the balance between underfitting and overfitting.

```
for i in range(2,16):  #power of 1 is already there
    colname = 'x_%d'%i      #new var will be x_power
    data[colname] = data['x']**i
data.head()
```

	x	y	x_2	x_3	x_4	x_5	x_6	x_7
0	0.000000	0.199738	0.000000	0.000000	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	0.017453	0.124744	0.000305	0.000005	9.279177e-08	1.619522e-09	2.826599e-11	4.933346e-13
2	0.034907	-0.196911	0.001218	0.000043	1.484668e-06	5.182470e-08	1.809023e-09	6.314683e-11
3	0.052360	0.051078	0.002742	0.000144	7.516134e-06	3.935438e-07	2.060591e-08	1.078923e-09
4	0.069813	0.162957	0.004874	0.000340	2.375469e-05	1.658390e-06	1.157775e-07	8.082794e-09

What This Code Does

- **Generates Higher-Degree Polynomial Features**: Iterates over the range **2 to 15**, computing polynomial terms (x^2, x^3, ..., x^1).
- **Dynamically Creates Column Names**: New feature names are automatically generated in the format **`x_2, x_3, ..., x_15`**.
- **Expands the Dataset**: Each polynomial-transformed feature is stored

11.2.2 Splitting the Data

Next, we will split the data into **training and testing sets**. As we've learned, models tend to **overfit** when trained on a small dataset.

To intentionally create an **overfitting scenario**, we will: - Use **only 10% of the data for training**. - Reserve **90% of the data for testing**.

This is **not a typical train-test split** but is deliberately done to **demonstrate overfitting**, where the model performs well on the training data but generalizes poorly to unseen data.

```
from sklearn.model_selection import train_test_split
train, test = train_test_split(data, test_size=0.9)
```

```
print('Number of observations in the training data:', len(train))
print('Number of observations in the test data:',len(test))
```

```
Number of observations in the training data: 36
Number of observations in the test data: 324
```

11.2.3 Splitting the target and features

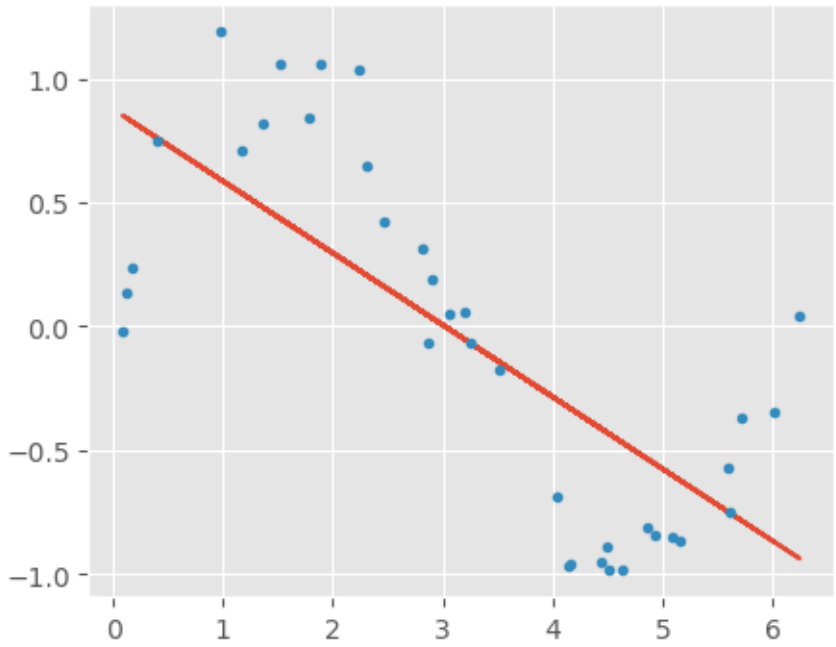
```
X_train = train.drop('y', axis=1).values
y_train = train.y.values
X_test = test.drop('y', axis=1).values
y_test = test.y.values
```

11.2.4 Building Models

11.2.4.1 Building a linear model with only 1 predictor `x`

```
# Linear regression with one feature
independent_variable_train = X_train[:, 0:1]

linreg = LinearRegression()
linreg.fit(independent_variable_train, y_train)
y_train_pred = linreg.predict(independent_variable_train)
rss_train = sum((y_train_pred-y_train)**2)/X_train.shape[0]

independent_variable_test = X_test[:, 0:1]
y_test_pred = linreg.predict(independent_variable_test)
rss_test = sum((y_test_pred-y_test)**2)/X_test.shape[0]

print("Training Error", rss_train)
print("Testing Error", rss_test)

plt.plot(X_train[:, 0:1], y_train_pred)
plt.plot(X_train[:, 0:1], y_train, '.');
```

```
Training Error 0.22398220582126424
Testing Error 0.22151086120574928
```



11.2.4.2 Building a linear regression model with three features `x, x_2, x_3`

```
independent_variable_train = X_train[:, 0:3]
independent_variable_train[:3]
```

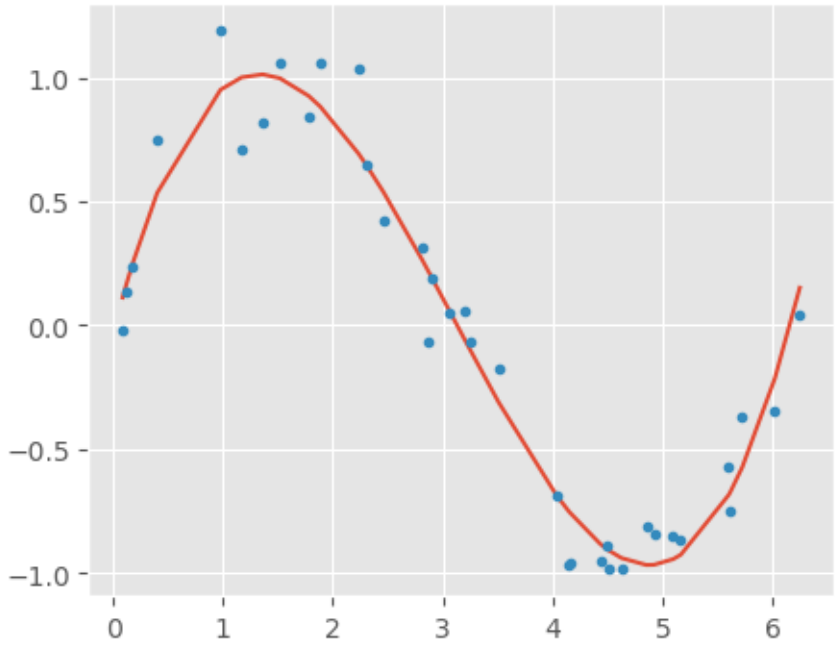
```
array([[ 1.36135682,  1.85329238,  2.52299222],
       [ 2.30383461,  5.30765392, 12.22795682],
       [ 1.51843645,  2.30564925,  3.50098186]])
```

```
def sort_xy(x,y):
    idx = np.argsort(x)
    x2,y2= x[idx] ,y[idx]
    return x2,y2
```

```
# Linear regression with 3 features

linreg = LinearRegression()

linreg.fit(independent_variable_train, y_train)
y_train_pred = linreg.predict(independent_variable_train)
```

```
rss_train = sum((y_train_pred-y_train)**2)/X_train.shape[0]

independent_variable_test = X_test[:, 0:3]
y_test_pred = linreg.predict(independent_variable_test)
rss_test = sum((y_test_pred-y_test)**2)/X_test.shape[0]

print("Training Error", rss_train)
print("Testing Error", rss_test)

plt.plot(*sort_xy(X_train[:, 0], y_train_pred))
plt.plot(X_train[:, 0], y_train, '.');
```

```
Training Error 0.02167114498970705
Testing Error 0.028159311299747036
```



Let's define a helper function that dynamically builds and trains a linear regression model based on a specified number of features. It allows for flexibility in selecting features and automates the process for multiple models.

```
# Define a function which will fit linear vregression model, plot the results, and return the
def linear_regression(train_x, train_y, test_x, test_y, features, models_to_plot):
```

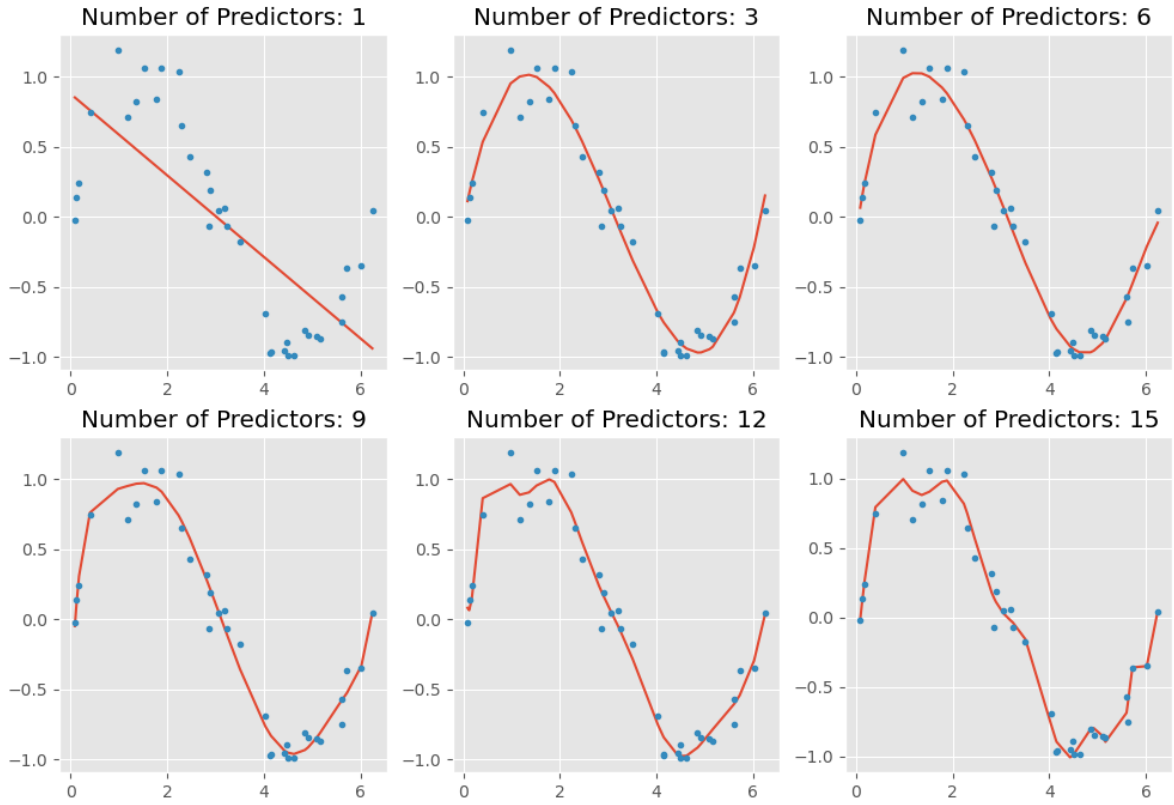
```
    #fit the model
    linreg = LinearRegression()
    linreg.fit(train_x, train_y)
    train_y_pred = linreg.predict(train_x)
    test_y_pred = linreg.predict(test_x)

    #check if a plot is to be made for the entered features
    if features in models_to_plot:
        plt.subplot(models_to_plot[features])
        # plt_tight_layout()
        plt.plot(*sort_xy(train_x[:, 0], train_y_pred))
        plt.plot(train_x[:, 0], train_y, '.')

        plt.title('Number of Predictors: %d'%features)

    #return the result in pre-defined format
    rss_train = sum((train_y_pred - train_y)**2)/train_x.shape[0]
    ret = [rss_train]

    rss_test = sum((test_y_pred - test_y)**2)/test_x.shape[0]
    ret.extend([rss_test])

    ret.extend([linreg.intercept_])
    ret.extend(linreg.coef_)

    return ret
```

```
#initialize a dataframe to store the results:
col = ['mrss_train', 'mrss_test', 'intercept'] + ['coef_VaR_%d'%i for i in range(1, 16)]
ind = ['Number_of_variable_%d'%i for i in range(1, 16)]
coef_matrix_simple = pd.DataFrame(index=ind, columns=col)
```

```
# Define the number of features for which a plot is required:
models_to_plot = {1:231, 3:232, 6:233, 9:234, 12:235, 15:236}
```

```
import matplotlib.pyplot as plt
# Iterate through all powers and store the results in a matrix form
plt.figure(figsize = (12, 8))
for i in range(1, 16):
    train_x = X_train[:, 0:i]
    train_y = y_train
    test_x = X_test[:, 0:i]
```

```
    test_y = y_test

    coef_matrix_simple.iloc[i-1, 0:i+3] = linear_regression(train_x, train_y, test_x, test_y
```



Key Takeaways:

As we increase the number of features (higher-degree polynomial terms), we observe the following:
- The model becomes **more flexible**, capturing intricate patterns in the training data.
- The curve becomes **increasingly wavy and complex**, adapting too closely to the data points.
- This results in **overfitting**, where the model performs well on the training set but fails to generalize to unseen data.

Overfitting occurs because the model learns **noise** instead of the true underlying pattern, leading to poor performance on new data.

To better understand this phenomenon, let's:
- **Evaluate model performance** on both the training and test sets.

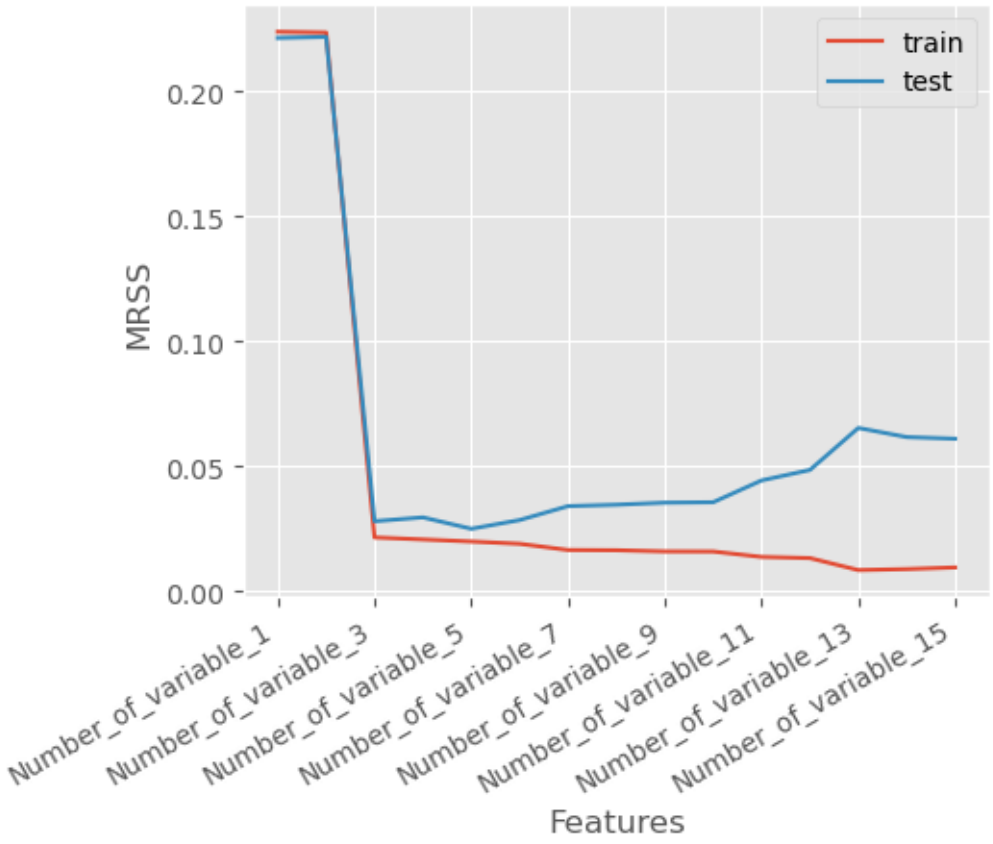
- **Output the model coefficients** to analyze how feature coefficients changes with increasing complexity.

```
# Set the display format to be scientific for ease of analysis
pd.options.display.float_format = '{:,.2g}'.format
coef_matrix_simple
```

	mrss_train	mrss_test	intercept	coef_VaR_1	coef_VaR_2	coef_VaR_3
Number_of_variable_1	0.22	0.22	0.88	-0.29	NaN	NaN
Number_of_variable_2	0.22	0.22	0.84	-0.25	-0.0057	NaN
Number_of_variable_3	0.022	0.028	-0.032	1.7	-0.83	0.089
Number_of_variable_4	0.021	0.03	-0.09	2	-1	0.14
Number_of_variable_5	0.02	0.025	-0.019	1.5	-0.48	-0.092
Number_of_variable_6	0.019	0.029	-0.13	2.4	-1.9	0.77
Number_of_variable_7	0.017	0.034	-0.37	4.7	-6.5	4.7
Number_of_variable_8	0.017	0.035	-0.42	5.3	-8	6.4
Number_of_variable_9	0.016	0.036	-0.57	7.1	-14	16
Number_of_variable_10	0.016	0.036	-0.51	6.2	-11	9.7
Number_of_variable_11	0.014	0.044	0.17	-4	36	-86
Number_of_variable_12	0.013	0.049	0.54	-10	67	-1.6e+02
Number_of_variable_13	0.0086	0.065	-0.56	9.9	-56	2e+02
Number_of_variable_14	0.009	0.062	-0.075	0.62	7	-5.1
Number_of_variable_15	0.0097	0.061	-0.3	3.4	-0.93	-2

Let's plot the training error versus the test error below and identify the overfitting

```
coef_matrix_simple[['mrss_train', 'mrss_test']].plot()
ax = plt.gca()
plt.xlabel('Features')
plt.ylabel('MRSS')
plt.setp(ax.get_xticklabels(), rotation=30, horizontalalignment='right')
plt.legend(['train', 'test']);
```



11.2.5 Overfitting Indicated by Training and Test MRSS Trends

As observed in the plot:
- The **Training Mean Residual Sum of Squares (MRSS)** consistently decreases as the number of features increases.
- However, after a certain point, the **Test MRSS starts to rise**, indicating that the model is no longer generalizing well to unseen data.

This trend suggests that while adding more features helps the model fit the **training data** better, it also causes the model to **memorize noise**, leading to poor performance on the test set.

This is a classic sign of **overfitting**, where the model captures excessive complexity in the data rather than the true underlying pattern.

Next, let's mitigate the overfitting issue using regularization

11.3 Regularization: Combating Overfitting

Regularization is a technique that **modifies the loss function** by adding a penalty term to control model complexity.
This helps **prevent overfitting** by discouraging large coefficients in the model.

11.3.1 Regularized Loss Function

The regularized loss function is given by:

$L_{reg}(\beta) = L(\beta) + \alpha R(\beta)$

where: - $L(\beta)$ is the original loss function (e.g., Mean Squared Error in linear regression).
- $R(\beta)$ is the **regularization term**, which penalizes large coefficients.
- α is a **hyperparameter** that controls the strength of regularization.

11.3.2 Regularization Does Not Penalize the Intercept

- The **intercept (bias term)** captures the **baseline mean** of the target variable.
- Penalizing the intercept would **shift predictions** incorrectly instead of controlling complexity.
- Thus, regularization **only applies to feature coefficients**, not the intercept.

11.3.3 Types of Regularization

1. **L1 Regularization (Lasso Regression)**: Encourages sparsity by driving some coefficients to zero.
2. **L2 Regularization (Ridge Regression)**: Shrinks coefficients but keeps all of them nonzero.
3. **Elastic Net**: A combination of both L1 and L2 regularization.

By applying **regularization**, we obtain models that balance **bias-variance tradeoff**, leading to **better generalization**.

11.3.4 Why Is Feature Scaling Required in Regularization?

11.3.4.1 The Effect of Feature Magnitudes on Regularization

Regularization techniques such as **Lasso (L1)**, **Ridge (L2)**, and **Elastic Net** apply penalties to the model's coefficients. However, when features have vastly different scales, **regularization disproportionately affects certain features**, leading to:

- **Uneven shrinkage** of coefficients, causing instability in the model.
- **Incorrect feature importance interpretation**, as some features dominate due to their larger numerical scale.
- **Suboptimal performance**, since regularization penalizes large coefficients more, even if they belong to more informative features.

11.3.4.2 Example: The Need for Feature Scaling

Imagine a dataset with two features:

- **Feature 1**: Number of bedrooms (range: 1-5).
- **Feature 2**: House area in square feet (range: 500-5000).

Since house area has much larger values, the model assigns smaller coefficients to compensate, making regularization unfairly **biased toward smaller-scale features**.

11.3.4.3 How to Scale Features for Regularization

To ensure fair treatment of all features, apply **feature scaling** before training a regularized model:

11.3.4.3.1 Standardization (Recommended)

$$x_{\text{scaled}} = \frac{x - \mu}{\sigma}$$

- Centers the data around **zero** with unit variance. - Used in **Lasso, Ridge, and Elastic Net**.

11.3.4.3.2 Min-Max Scaling

$$x_{\text{scaled}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

- Scales features to a fixed **[0, 1]** range. - Common for **neural networks** but less effective for regularization.

Let's use `StandardScaler` to scale the features

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Feature scaling on X_train
X_train_std = scaler.fit_transform(X_train)
columns = data.drop('y', axis=1).columns
X_train_std = pd.DataFrame(X_train_std, columns = columns)
X_train_std.head()
# Feature scaling on X_test
X_test_std = scaler.transform(X_test)
X_test_std = pd.DataFrame(X_test_std, columns = columns)
X_test_std.head()
```

	x	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_10	x_11	x_12	x_13	x_14
0	-0.96	-1	-0.89	-0.78	-0.69	-0.62	-0.57	-0.52	-0.48	-0.45	-0.43	-0.4	-0.38	-0.37
1	-0.061	-0.34	-0.49	-0.55	-0.57	-0.56	-0.53	-0.5	-0.47	-0.45	-0.42	-0.4	-0.38	-0.37
2	-1.8	-1.2	-0.95	-0.8	-0.7	-0.62	-0.57	-0.52	-0.48	-0.45	-0.43	-0.4	-0.38	-0.37
3	0.21	-0.051	-0.24	-0.36	-0.43	-0.46	-0.47	-0.46	-0.45	-0.43	-0.41	-0.4	-0.38	-0.36
4	-1.7	-1.2	-0.95	-0.8	-0.7	-0.62	-0.57	-0.52	-0.48	-0.45	-0.43	-0.4	-0.38	-0.37

In the next section, we will explore **different types of regularization techniques** and see how they help in preventing overfitting.

11.3.5 Ridge Regression: L2 Regularization

Ridge regression is a type of **linear regression** that incorporates **L2 regularization** to prevent overfitting by **penalizing large coefficients**.

11.3.5.1 Ridge Regression Loss Function

The regularized loss function for Ridge regression is given by:

$$L_{\text{Ridge}}(\beta) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \beta^\top x_i|^2 + \alpha\sum_{j=1}^{J}\beta_j^2$$

where:

- y_i is the true target value for observation i.
- x_i is the feature vector for observation i.
- β is the vector of model coefficients.
- α is the regularization parameter, which controls the penalty strength.
- $\sum_{j=1}^{J}\beta_j^2$ is the L2 norm (sum of squared coefficients).

Note that j starts from 1, excluding the intercept from regularization.

The penalty term in Ridge regression,

$$\sum_{j=1}^{J}\beta_j^2 = ||\beta||_2^2$$

is the **squared L2 norm** of the coefficient vector β.
Minimizing this norm ensures that the model coefficients remain **small and stable**, reducing sensitivity to variations in the data.

Let's build a Ridge Regression model using scikit-learn, The `alpha` parameter controls the strength of the regularization:

```
from sklearn.linear_model import Ridge
# defining a function which will fit ridge regression model, plot the results, and return the
def ridge_regression(train_x, train_y, test_x, test_y, alpha, models_to_plot={}):

    #fit the model
    ridgereg = Ridge(alpha=alpha)
    ridgereg.fit(train_x, train_y)
    train_y_pred = ridgereg.predict(train_x)
    test_y_pred = ridgereg.predict(test_x)

    #check if a plot is to be made for the entered alpha
    if alpha in models_to_plot:
        plt.subplot(models_to_plot[alpha])
        # plt_tight_layout()
```

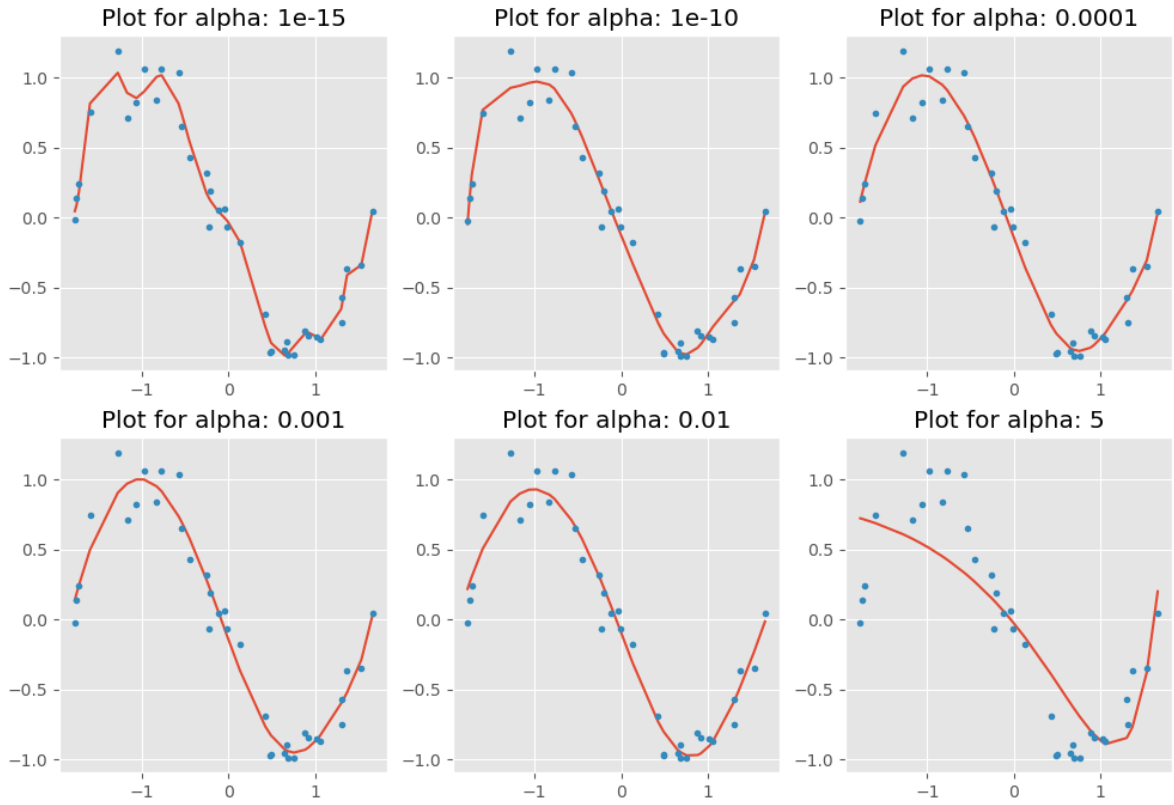
```
        plt.plot(*sort_xy(train_x.values[:, 0], train_y_pred))
        plt.plot(train_x.values[:, 0], train_y, '.')

        plt.title('Plot for alpha: %.3g'%alpha)

    #return the result in pre-defined format
    mrss_train = sum((train_y_pred - train_y)**2)/train_x.shape[0]
    ret = [mrss_train]

    mrss_test = sum((test_y_pred - test_y)**2)/test_x.shape[0]
    ret.extend([mrss_test])

    ret.extend([ridgereg.intercept_])
    ret.extend(ridgereg.coef_)

    return ret
```

Let's experiment with different values of alpha in Ridge Regression and observe how it affects the model's coefficients and performance.

```
#initialize a dataframe to store the coefficient:
alpha_ridge = [1e-15, 1e-10, 1e-8, 1e-4, 1e-3,1e-2, 1, 5, 10, 20]
col = ['mrss_train', 'mrss_test', 'intercept'] + ['coef_VaR_%d'%i for i in range(1, 16)]
ind = ['alpha_%.2g'%alpha_ridge[i] for i in range(0, 10)]
coef_matrix_ridge = pd.DataFrame(index=ind, columns=col)
```

```
# Define the number of features for which a plot is required:
models_to_plot = {1e-15:231, 1e-10:232, 1e-4:233, 1e-3:234, 1e-2:235, 5:236}
```

```
#Iterate over the 10 alpha values:
plt.figure(figsize=(12, 8))
for i in range(10):
    coef_matrix_ridge.iloc[i,] = ridge_regression(X_train_std, train_y, X_test_std, test_y, a
```



As we can observe, when increasing alpha, the model becomes simpler, with coefficients shrinking more aggressively due to stronger regularization. This reduces the risk of overfitting but may lead to underfitting if alpha is set too high.

Next, let's output the training error versus the test error and examine how the feature coefficients change with different alpha values.

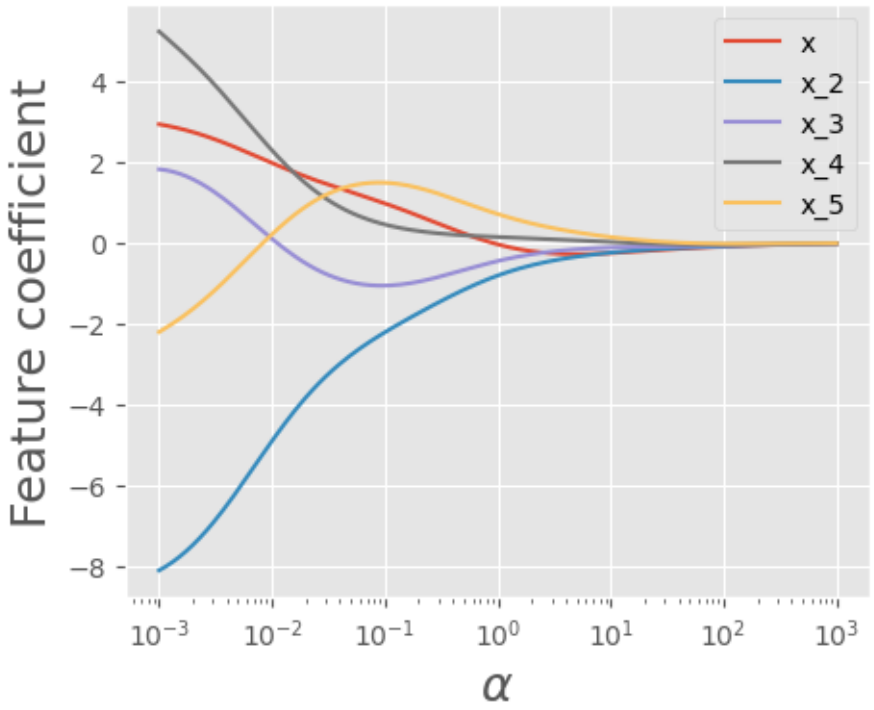
```
#Set the display format to be scientific for ease of analysis
pd.options.display.float_format = '{:,.2g}'.format
coef_matrix_ridge
```

	mrss_train	mrss_test	intercept	coef_VaR_1	coef_VaR_2	coef_VaR_3	coef_VaR_
alpha_1e-15	0.0092	0.059	-0.072	-2.8	1.9e+02	-1.3e+03	-6e+03
alpha_1e-10	0.016	0.036	-0.072	12	-1.3e+02	7.1e+02	-1.8e+03
alpha_1e-08	0.017	0.034	-0.072	8.1	-70	2.8e+02	-5.6e+02
alpha_0.0001	0.019	0.024	-0.072	2.9	-8.2	4.5	-0.022
alpha_0.001	0.019	0.023	-0.072	2.5	-5.6	-0.5	1.3
alpha_0.01	0.022	0.024	-0.072	1.9	-4	-1.3	0.73

	mrss_train	mrss_test	intercept	coef_VaR_1	coef_VaR_2	coef_VaR_3	coef_VaR_
alpha_1	0.089	0.093	-0.072	0.08	-0.61	-0.48	-0.22
alpha_5	0.12	0.13	-0.072	-0.17	-0.3	-0.24	-0.14
alpha_10	0.14	0.14	-0.072	-0.19	-0.24	-0.18	-0.12
alpha_20	0.16	0.17	-0.072	-0.18	-0.19	-0.15	-0.097

To better observe the pattern, let's visualize how the coefficients change as we increase α

```
def plot_ridge_reg_coeff(train_x):
    alphas = np.logspace(3,-3,200)
    coefs = []
    #X_train_std, train_y
    for a in alphas:
        ridge = Ridge(alpha = a)
        ridge.fit(train_x, train_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(r'$\alpha$')
    plt.ylabel('Feature coefficient')
    plt.legend(train_x.columns );
plot_ridge_reg_coeff(X_train_std.iloc[:,:5])
plt.savefig("test.png")
```



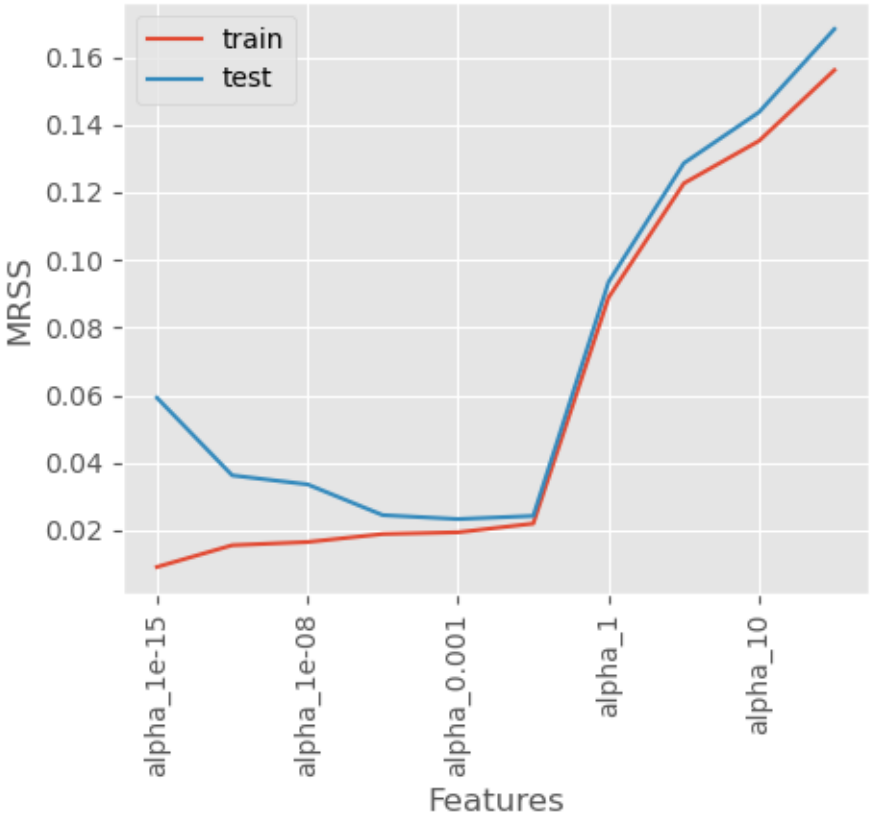
As we can see, as α increases, the coefficients become smaller and approach zero. Now, let's examine the number of zero coefficients.

```
coef_matrix_ridge.apply(lambda x: sum(x.values==0),axis=1)
```

```
alpha_1e-15     0
alpha_1e-10     0
alpha_1e-08     0
alpha_0.0001    0
alpha_0.001     0
alpha_0.01      0
alpha_1         0
alpha_5         0
alpha_10        0
alpha_20        0
dtype: int32
```

Let's plot how the test error and training error change as we increase α

```
coef_matrix_ridge[['mrss_train', 'mrss_test']].plot()
plt.xlabel('Features')
plt.ylabel('MRSS')
plt.xticks(rotation=90)
plt.legend(['train', 'test']);
```



As we can observe, as λ increases beyond a certain value, both the training MRSS and test MRSS begin to rise, indicating that the model starts underfitting.

11.3.6 Lasso Regression: L1 Regularization

LASSO stands for **Least Absolute Shrinkage and Selection Operator**. There are two key aspects in this name:
- **"Absolute"** refers to the use of the absolute values of the coefficients in the penalty term.
- **"Selection"** highlights LASSO's ability to shrink some coefficients to exactly zero, effectively performing **feature selection**.

Lasso regression performs **L1 regularization**, which helps prevent overfitting by penalizing large coefficients and enforcing sparsity in the model.

11.3.6.1 Lasso Regression Loss Function

In standard **linear regression**, the loss function is the **Mean Squared Error (MSE)**:

$$L_{\text{MSE}}(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta^\top x_i)^2$$

LASSO modifies this by adding an **L1 regularization** penalty, leading to the following **regularized loss function**:

$$L_{\text{Lasso}}(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta^\top x_i)^2 + \alpha\sum_{j=1}^{J}|\beta_j|$$

where:

- y_i is the true target value for observation i.
- x_i is the feature vector for observation i.
- β is the vector of model coefficients.
- α is the regularization parameter, which controls the penalty strength.
- $\sum_{j=1}^{J}|\beta_j|$ is the $\mathbf{L_1}$ norm (sum of absolute values of coefficients).

The penalty term in Lasso regression,

$$\sum_{j=1}^{J}|\beta_j| = ||\beta||_1$$

is the **L1 norm** of the coefficient vector (β).
Minimizing this norm encourages **sparsity**, meaning some coefficients become exactly zero, leading to an automatically selected subset of features.

Next, let's build a Lasso Regression model. Similar to Ridge regression, we will explore a range of values for the regularization parameter alpha.

```
from sklearn.linear_model import Lasso
alpha_lasso = [1e-15, 1e-10, 1e-8, 1e-5,1e-4, 1e-3,1e-2, 0.1, 1, 5]
```

```
# Defining a function which will fit lasso regression model, plot the results, and return the
def lasso_regression(train_x, train_y, test_x, test_y, alpha, models_to_plot={}):

    #fit the model
    if alpha == 0:
        lassoreg = LinearRegression()
    else:
        lassoreg = Lasso(alpha=alpha, max_iter=200000000, tol=0.01)
    lassoreg.fit(train_x, train_y)
    train_y_pred = lassoreg.predict(train_x)
    test_y_pred = lassoreg.predict(test_x)



    #check if a plot is to be made for the entered alpha
    if alpha in models_to_plot:
        plt.subplot(models_to_plot[alpha])
        # plt_tight_layout()
        plt.plot(*sort_xy(train_x.values[:, 0], train_y_pred))
        plt.plot(train_x.values[:, 0:1], train_y, '.')

        plt.title('Plot for alpha: %.3g'%alpha)

    #return the result in pre-defined format
    mrss_train = sum((train_y_pred - train_y)**2)/train_x.shape[0]
    ret = [mrss_train]

    mrss_test = sum((test_y_pred - test_y)**2)/test_x.shape[0]
    ret.extend([mrss_test])

    ret.extend([lassoreg.intercept_])
    ret.extend(lassoreg.coef_)

    return ret
```

```
#initialize a dataframe to store the coefficient:
col = ['mrss_train', 'mrss_test', 'intercept'] + ['coef_VaR_%d'%i for i in range(1, 16)]
ind = ['alpha_%.2g'%alpha_lasso[i] for i in range(0, 10)]
coef_matrix_lasso = pd.DataFrame(index=ind, columns=col)
```

```
# Define the number of features for which a plot is required:
models_to_plot = {1e-10:231, 1e-5:232,1e-4:233, 1e-3:234, 1e-2:235, 0.1:236}
```

```
models_to_plot
```

```
{1e-10: 231, 1e-05: 232, 0.0001: 233, 0.001: 234, 0.01: 235, 0.1: 236}
```

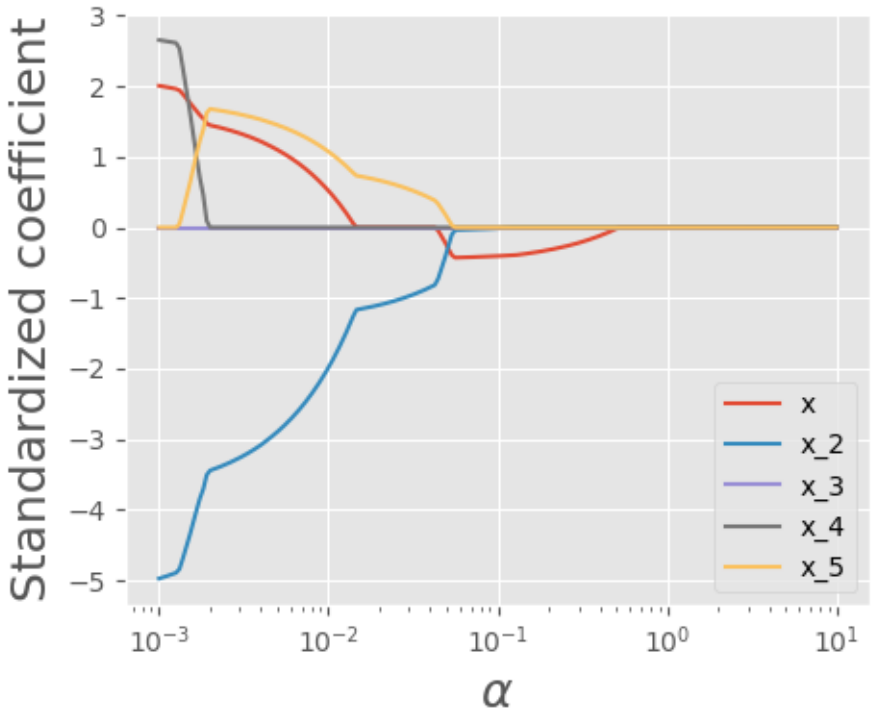
```
#Iterate over the 10 alpha values:
plt.figure(figsize=(12, 8))
for i in range(10):
    coef_matrix_lasso.iloc[i,] = lasso_regression(X_train_std, train_y, X_test_std, test_y, a
```

```
#Set the display format to be scientific for ease of analysis
pd.options.display.float_format = '{:,.2g}'.format
coef_matrix_lasso
```

	mrss_train	mrss_test	intercept	coef_VaR_1	coef_VaR_2	coef_VaR_3	coef_VaR_
alpha_1e-15	0.019	0.024	-0.072	2.8	-6.9	0.83	1.5
alpha_1e-10	0.019	0.024	-0.072	2.8	-6.9	0.83	1.5
alpha_1e-08	0.019	0.024	-0.072	2.8	-6.9	0.83	1.5
alpha_1e-05	0.019	0.024	-0.072	2.8	-6.8	0.73	1.7
alpha_0.0001	0.02	0.024	-0.072	2.7	-6.5	0	2.8
alpha_0.001	0.024	0.026	-0.072	2	-4.5	-0	0
alpha_0.01	0.084	0.088	-0.072	0.27	-1.3	-0	-0
alpha_0.1	0.19	0.2	-0.072	-0.22	-0.27	-0	-0
alpha_1	0.49	0.53	-0.072	-0	-0	-0	-0
alpha_5	0.49	0.53	-0.072	-0	-0	-0	-0

```
def plot_lasso_reg_coeff(train_x):
    alphas = np.logspace(1,-3,200)
    coefs = []
    #X_train_std, train_y
    for a in alphas:
        laso = Lasso(alpha=a, max_iter = 100000)
        laso.fit(train_x, train_y)
        coefs.append(laso.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(r'$\alpha$')
    plt.ylabel('Standardized coefficient')
    plt.legend(train_x.columns );
plot_lasso_reg_coeff(X_train_std.iloc[:,:5])
plt.savefig("test1.png")
```

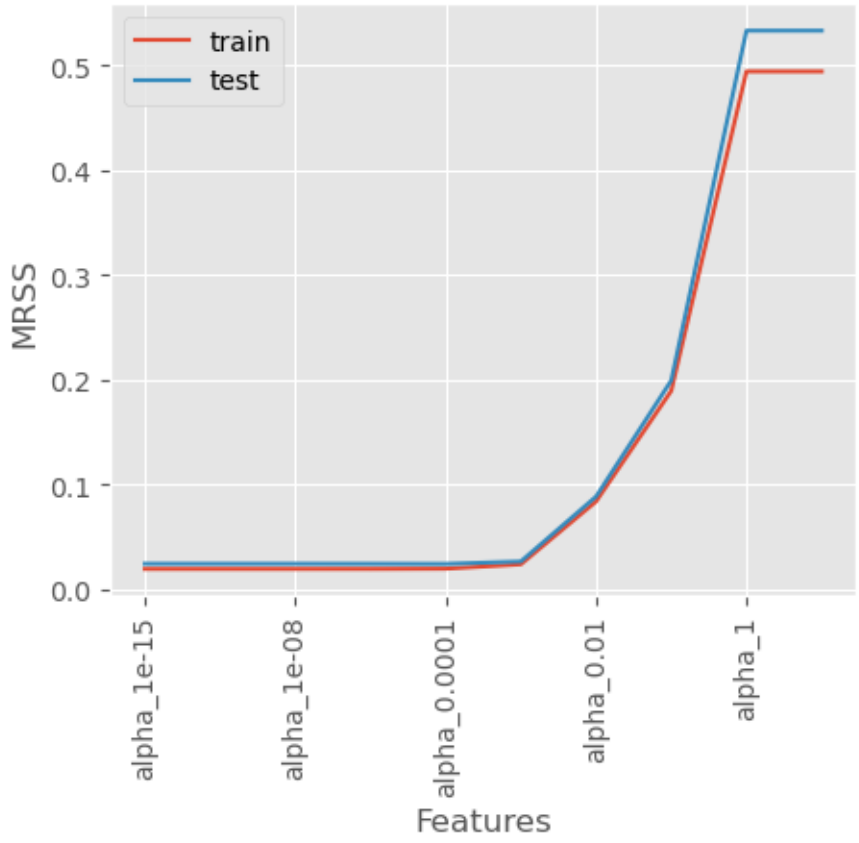



```
coef_matrix_lasso.apply(lambda x: sum(x.values==0),axis=1)
```

```
alpha_1e-15      0
alpha_1e-10      0
alpha_1e-08      0
alpha_1e-05      2
alpha_0.0001     8
alpha_0.001     11
alpha_0.01      12
alpha_0.1       12
alpha_1         15
```

```
alpha_5          15
dtype: int32
```

```
coef_matrix_lasso[['mrss_train', 'mrss_test']].plot()
plt.xlabel('Features')
plt.ylabel('MRSS')
plt.xticks(rotation=90)
plt.legend(['train', 'test'])
```



Effect of alpha in Lasso Regression - Small alpha (close to 0): The penalty is minimal, and Lasso behaves like ordinary linear regression. - **Moderate alpha:** Some coefficients shrink, and some become exactly zero, performing feature selection. - **Large alpha:** Many coefficients become zero, leading to a very simple model (potentially underfitting).

11.3.7 Elastic Net Regression: Combining L1 and L2 Regularization

11.3.7.1 Mathematical Formulation*

Elastic Net regression combines both **L1 (Lasso) and L2 (Ridge) penalties**, balancing feature selection and coefficient shrinkage. The **regularized loss function** for Elastic Net is given by:

$$L_{\text{ElasticNet}}(\beta) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \beta^\top x_i)^2 + \alpha\left((1-\rho)\frac{1}{2}\sum_{j=1}^{p}\beta_j^2 + \rho\sum_{j=1}^{p}|\beta_j|\right)$$

where:

- y_i is the true target value for observation i.
- x_i is the feature vector for observation i.
- β is the vector of model coefficients.
- α is the **regularization strength** parameter in scikit-learn.
- ρ is the **l1_ratio** parameter in scikit-learn, controlling the mix of L1 and L2 penalties.
- $\sum_{j=1}^{p}|\beta_j|$ is the **L1 norm**, enforcing sparsity.
- $\sum_{j=1}^{p}\beta_j^2$ is the **L2 norm**, ensuring coefficient stability.

11.3.7.2 Elastic Net Special Cases

- When **l1_ratio = 0**, Elastic Net behaves like **Ridge Regression (L2 regularization)**.
- When **l1_ratio = 1**, Elastic Net behaves like **Lasso Regression (L1 regularization)**.
- When **0 < l1_ratio < 1**, Elastic Net **balances feature selection (L1) and coefficient shrinkage (L2)**.

Now, let's implement an **Elastic Net Regression model** using `scikit-learn` and explore how different values of `alpha` and `l1_ratio` affect the model.

```
from sklearn.linear_model import ElasticNet
alpha_lasso = [1e-15, 1e-10, 1e-8, 1e-5,1e-4, 1e-3,1e-2, 0.1, 1, 5]
l1_ratio=0.5
```

```
# Defining a function which will fit lasso regression model, plot the results, and return the
def elasticnet_regression(train_x, train_y, test_x, test_y, alpha, models_to_plot={}):

    #fit the model
    if alpha == 0:
        model = LinearRegression()
```

```
    else:
        model = ElasticNet(alpha=alpha, max_iter=20000, l1_ratio=l1_ratio)
    model.fit(train_x, train_y)
    train_y_pred = model.predict(train_x)
    test_y_pred = model.predict(test_x)



    #check if a plot is to be made for the entered alpha
    if alpha in models_to_plot:
        plt.subplot(models_to_plot[alpha])
        # plt_tight_layout()
        plt.plot(*sort_xy(train_x.values[:, 0], train_y_pred))
        plt.plot(train_x.values[:, 0:1], train_y, '.')

        plt.title('Plot for alpha: %.3g'%alpha)

    #return the result in pre-defined format
    mrss_train = sum((train_y_pred - train_y)**2)/train_x.shape[0]
    ret = [mrss_train]

    mrss_test = sum((test_y_pred - test_y)**2)/test_x.shape[0]
    ret.extend([mrss_test])

    ret.extend([model.intercept_])
    ret.extend(model.coef_)

    return ret
```

```
#initialize a dataframe to store the coefficient:
col = ['mrss_train', 'mrss_test', 'intercept'] + ['coef_VaR_%d'%i for i in range(1, 16)]
ind = ['alpha_%.2g'%alpha_lasso[i] for i in range(0, 10)]
coef_matrix_elasticnet = pd.DataFrame(index=ind, columns=col)
```

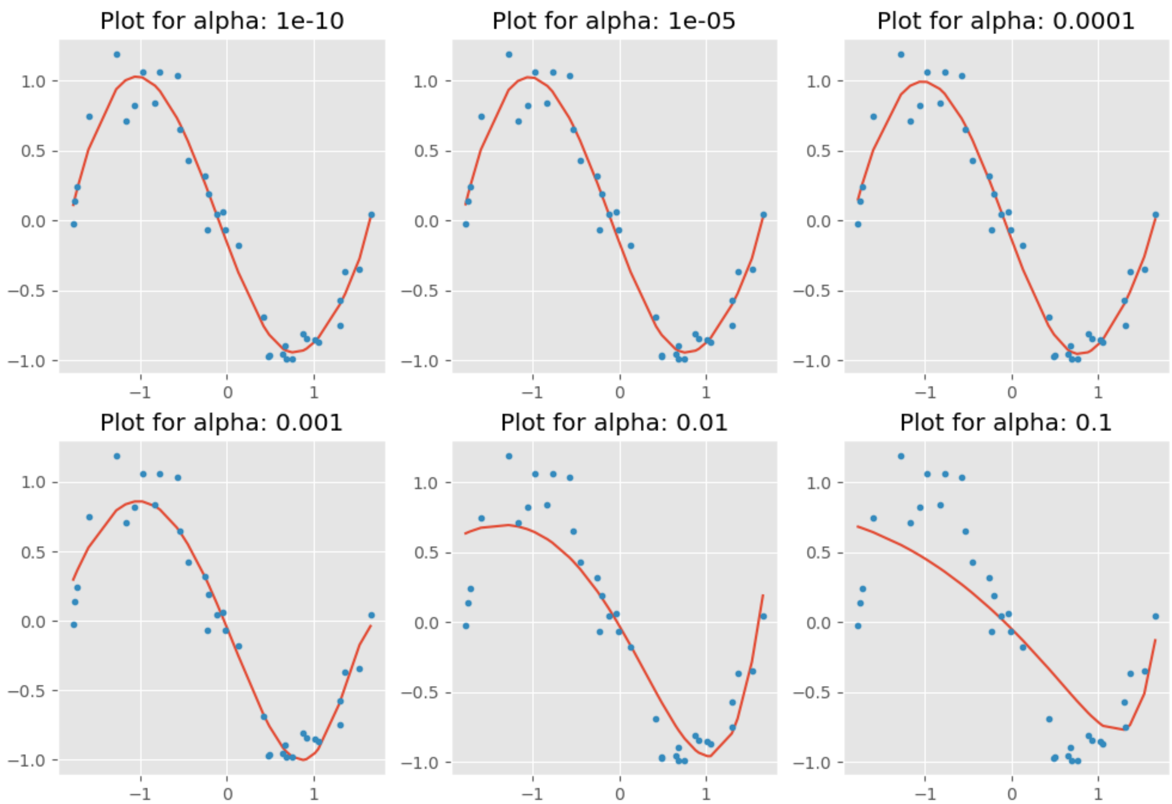
```
# Define the number of features for which a plot is required:
models_to_plot = {1e-10:231, 1e-5:232,1e-4:233, 1e-3:234, 1e-2:235, 0.1:236}
```

```
models_to_plot
```

```
{1e-10: 231, 1e-05: 232, 0.0001: 233, 0.001: 234, 0.01: 235, 0.1: 236}
```

```python
#Iterate over the 10 alpha values:
plt.figure(figsize=(12, 8))
for i in range(10):
    coef_matrix_elasticnet.iloc[i,] = elasticnet_regression(X_train_std, train_y, X_test_std
```

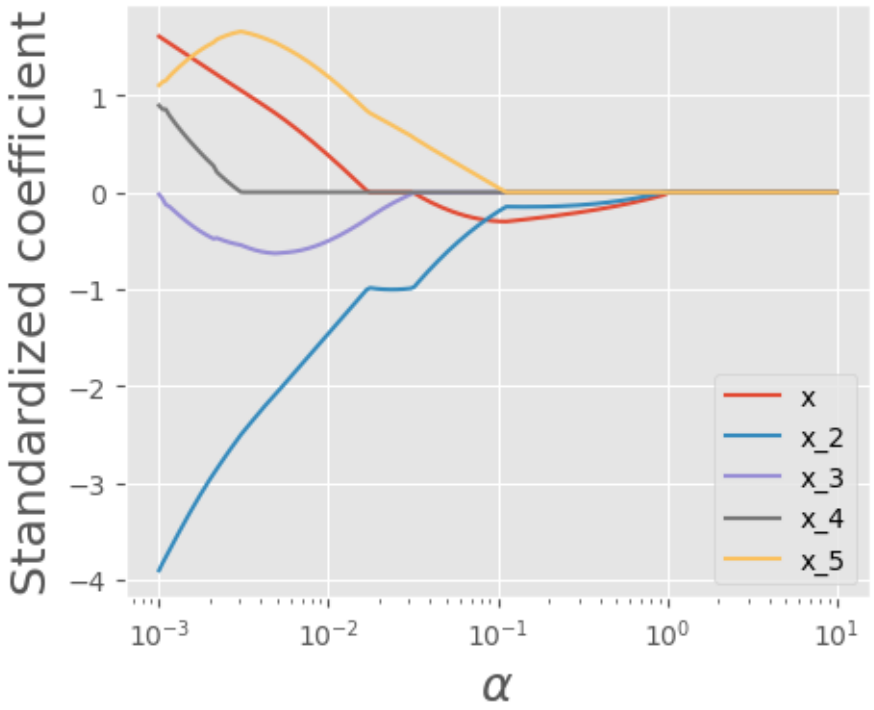
```
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\linear_model\_coordinate_d
  model = cd_fast.enet_coordinate_descent(
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\linear_model\_coordinate_d
  model = cd_fast.enet_coordinate_descent(
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\linear_model\_coordinate_d
  model = cd_fast.enet_coordinate_descent(
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\linear_model\_coordinate_d
  model = cd_fast.enet_coordinate_descent(
c:\Users\lsi8012\AppData\Local\anaconda3\Lib\site-packages\sklearn\linear_model\_coordinate_d
  model = cd_fast.enet_coordinate_descent(
```



```
#Set the display format to be scientific for ease of analysis
pd.options.display.float_format = '{:,.2g}'.format
coef_matrix_elasticnet
```

	mrss_train	mrss_test	intercept	coef_VaR_1	coef_VaR_2	coef_VaR_3	coef_VaR_
alpha_1e-15	0.019	0.024	-0.072	2.8	-6.9	0.83	1.5
alpha_1e-10	0.019	0.024	-0.072	2.8	-6.9	0.83	1.5
alpha_1e-08	0.019	0.024	-0.072	2.8	-6.9	0.83	1.5
alpha_1e-05	0.019	0.024	-0.072	2.7	-6.6	0.37	1.7
alpha_0.0001	0.02	0.023	-0.072	2.4	-5.5	-0.6	1.3
alpha_0.001	0.028	0.03	-0.072	1.6	-3.3	-0.6	0
alpha_0.01	0.076	0.081	-0.072	0.31	-1.1	-0.48	-0
alpha_0.1	0.15	0.16	-0.072	-0.19	-0.41	-0.0068	-0
alpha_1	0.48	0.52	-0.072	-0.013	-0	-0	-0
alpha_5	0.49	0.53	-0.072	-0	-0	-0	-0

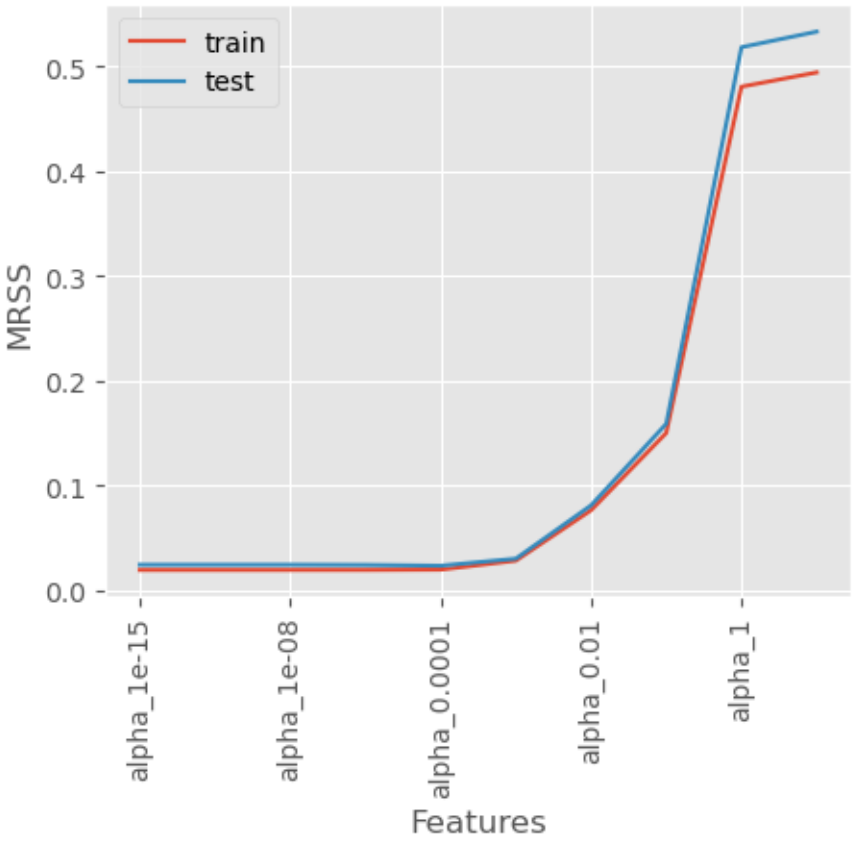
```
def plot_elasticnet_reg_coeff(train_x):
    alphas = np.logspace(1,-3,200)
    coefs = []
    #X_train_std, train_y
    for a in alphas:
        model = ElasticNet(alpha=a, max_iter=20000, l1_ratio=l1_ratio)
        model.fit(train_x, train_y)
        coefs.append(model.coef_)
    #Visualizing the 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(r'$\alpha$')
    plt.ylabel('Standardized coefficient')
    plt.legend(train_x.columns );
plot_elasticnet_reg_coeff(X_train_std.iloc[:,:5])
plt.savefig("test2.png")
```




```
coef_matrix_elasticnet.apply(lambda x: sum(x.values==0),axis=1)
```

```
alpha_1e-15      0
alpha_1e-10      0
alpha_1e-08      0
alpha_1e-05      0
alpha_0.0001     3
alpha_0.001      6
alpha_0.01       7
alpha_0.1        8
alpha_1         14
alpha_5         15
dtype: int32
```

```
coef_matrix_elasticnet[['mrss_train', 'mrss_test']].plot()
plt.xlabel('Features')
plt.ylabel('MRSS')
plt.xticks(rotation=90)
plt.legend(['train', 'test']);
```



ElasticNet is controlled by these key parameters:

- **alpha** (`float`, default=`1.0`):
  The regularization strength multiplier. Higher values increase regularization.
- **l1_ratio** (`float`, default=`0.5`):
  The mixing parameter between L1 and L2 penalties:
  - `l1_ratio = 0`: Pure Ridge regression
  - `l1_ratio = 1`: Pure Lasso regression
  - `0 < l1_ratio < 1`: ElasticNet with mixed penalties

11.3.8 `RidgeCV`, `LassoCV`, and `ElasticNetCV` in Scikit-Learn

In Scikit-Learn, `RidgeCV`, `LassoCV`, and `ElasticNetCV` are cross-validation (CV) versions of **Ridge**, **Lasso**, and **Elastic Net** regression models, respectively. These versions **auto-**

matically select the best regularization strength (`alpha`) by performing internal cross-validation.

11.3.8.1 Overview of RidgeCV, LassoCV, and ElasticNetCV**

Model	Regularization Type	Purpose	How Alpha is Chosen?
RidgeCV	L2 (Ridge)	Shrinks coefficients to handle overfitting, but keeps all features.	Uses cross-validation to select the best `alpha`.
LassoCV	L1 (Lasso)	Shrinks coefficients, but also **removes** some features by setting coefficients to zero.	Uses cross-validation to find the best `alpha`.
ElasticNetCV	L1 + L2 (Elastic Net)	Balances Ridge and Lasso.	Uses cross-validation to find the best `alpha` and `l1_ratio`.

11.3.8.2 How to Use `RidgeCV`, `LassoCV`, and`ElasticNetCV`

Each model automatically selects the optimal `alpha` value through internal cross-validation without using a loop through the `alpha` values

```
from sklearn.linear_model import LassoCV

alpha_lasso = [1e-15, 1e-10, 1e-8, 1e-5, 1e-4, 1e-3, 1e-2, 0.1, 1, 5]

# Initialize LassoCV with cross-validation
lasso_cv = LassoCV(alphas=alpha_lasso, cv=5, max_iter=2000000)

# Fit the model using training data
lasso_cv.fit(X_train_std, train_y)

# Make predictions
train_y_pred = lasso_cv.predict(X_train_std)
test_y_pred = lasso_cv.predict(X_test_std)
```

```
# Get the best alpha chosen by cross-validation
best_alpha = lasso_cv.alpha_
print(f"Best alpha selected by LassoCV: {best_alpha}")

# Check if a plot should be made for the selected alpha
if best_alpha in models_to_plot:
    plt.subplot(models_to_plot[best_alpha])
    plt.plot(*sort_xy(train_x[:, 0], train_y_pred))
    plt.plot(train_x[:, 0:1], train_y, '.', label="Actual Data")

    plt.title(f'Plot for alpha: {best_alpha:.3g}')
    plt.legend();
```

11.4 Reference

https://www.linkedin.com/pulse/tutorial-ridge-lasso-regression-subhajit-mondal/?trk=read_related_article-card_title

12 Feature Selection

12.1 Feature Selection

Feature selection is a crucial step in machine learning that helps improve model performance, reduce overfitting, and speed up training time by selecting the most relevant features from the dataset.

There are three main types of feature selection methods:

- Filter Methods (Based on statistical measures)
- Wrapper Methods (Based on model performance)
- Embedded Methods (Feature selection integrated within the model)

Why feature selection?

- Removes irrelevant or redundant features
- Reduces computational complexity
- Helps avoid the curse of dimensionality
- Improves model interpretability
- Enhances generalization by reducing overfitting

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

12.2 Filter Methods (Statistical approaches)

Filter methods evaluate the relevance of features based on statistical measures before training a model. These methods rank features according to their correlation with the target variable and select the most relevant ones.

12.2.1 Characteristics:

- **Independent of the model:** Feature selection is performed before training the model.
- **Fast and computationally efficient:** Since no model training is required, they are suitable for large datasets.
- **Prone to selecting irrelevant features:** They do not consider interactions between features.
- **Common techniques:**
  - Pearson correlation coefficient
  - Mutual information
  - Chi-square test
  - ANOVA (Analysis of Variance)
  - Variance thresholding

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

12.2.2 Methods

Method	Description	Python Function/Class	Notes
Variance Threshold	Removes low-variance numeric features	`VarianceThreshold()` (scikit-learn)	Only works on numerical features
Correlation Filter	Removes highly correlated features	Custom implementation + `df.corr()`	Requires manual analysis (pandas) or `correlation_threshold` methods
Chi-Square Test	Selects categorical features for classification	`chi2()` + `SelectKBest()`	For categorical targets, requires non-negative features
Mutual Information	Measures dependency for classification/regression	`mutual_info_classif()`/`regression`	Use with `SelectKBest()` or `SelectPercentile()`

```
from sklearn.feature_selection import VarianceThreshold, SelectKBest, f_regression

# Load the data
df = pd.read_csv('./Datasets/Credit.csv', index_col=0)

# Separate features and target
```

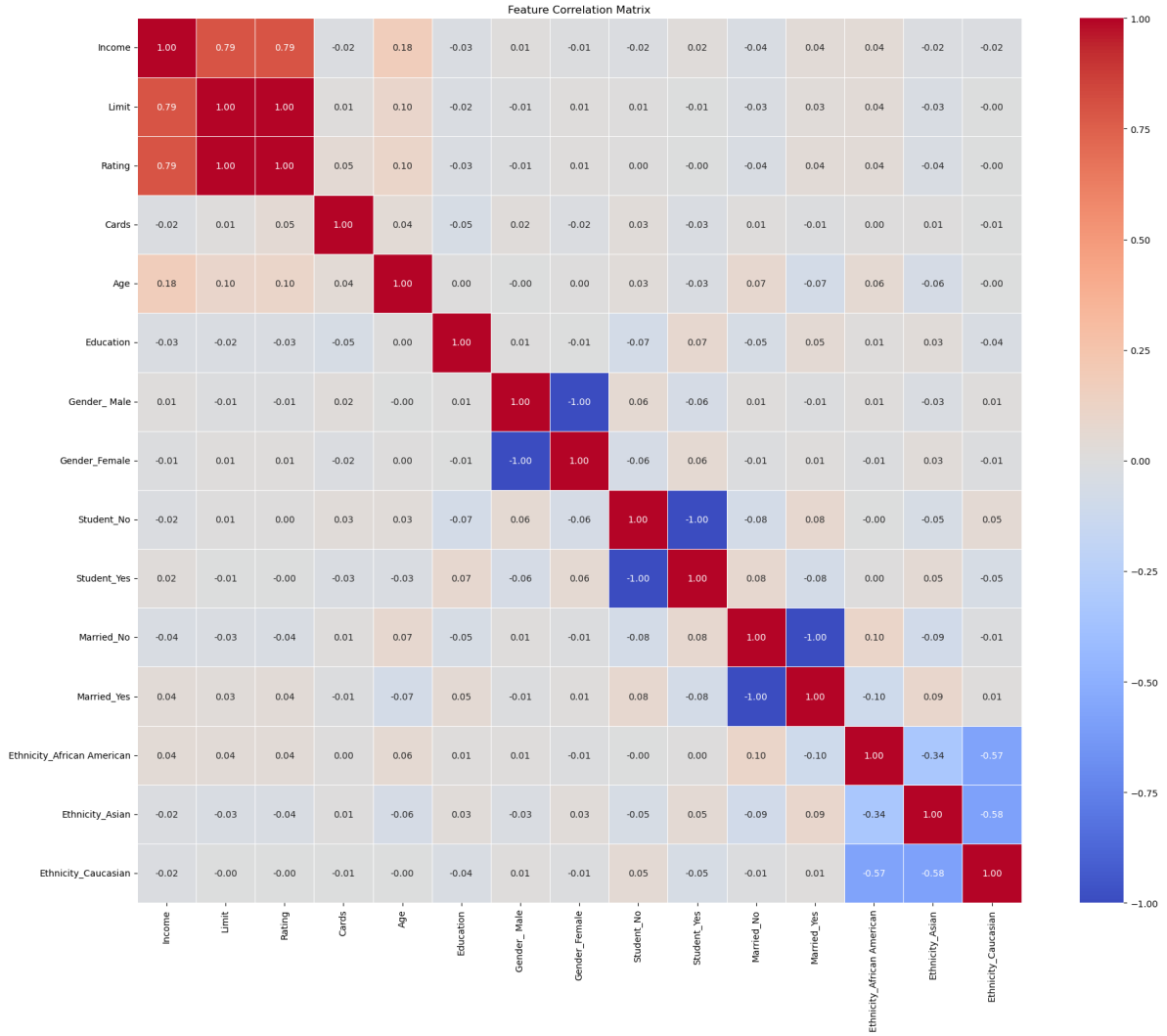
```
# Let's use 'Balance' as our target variable
X = df.drop('Balance', axis=1)
y = df['Balance']

# Preprocessing: Encode categorical variables
X_encoded = pd.get_dummies(X, columns=['Gender', 'Student', 'Married', 'Ethnicity'])

# 1. Variance Threshold Feature Selection
def variance_threshold_selection(X, threshold=0.1):
    # Create a VarianceThreshold selector
    selector = VarianceThreshold(threshold=threshold)

    # Fit and transform the data
    X_selected = selector.fit_transform(X)

    # Get the selected feature names
    selected_features = X.columns[selector.get_support()]

    return X_selected, selected_features

# Apply Variance Threshold
X_var_selected, var_selected_features = variance_threshold_selection(X_encoded)

print("1. Variance Threshold Feature Selection:")
print("Selected features:", list(var_selected_features))
print("Number of features reduced from", X_encoded.shape[1], "to", len(var_selected_features)
```

```
1. Variance Threshold Feature Selection:
Selected features: ['Income', 'Limit', 'Rating', 'Cards', 'Age', 'Education', 'Gender_ Male'
Number of features reduced from 15 to 13
```

```
# 2. Correlation-based Feature Removal
def remove_highly_correlated_features(X, threshold=0.8):
    # Compute the correlation matrix
    corr_matrix = X.corr().abs()

    # Create a mask of the upper triangle of the correlation matrix
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))

    # Find features to drop
    to_drop = [column for column in upper.columns if any(upper[column] > threshold)]
```

```
    # Drop highly correlated features
    X_reduced = X.drop(columns=to_drop)

    return X_reduced, to_drop

# Apply Correlation-based Feature Removal
X_corr_removed, dropped_features = remove_highly_correlated_features(X_encoded)

print("\n3. Correlation-based Feature Removal:")
print("Dropped features:", dropped_features)
print("Number of features reduced from", X_encoded.shape[1], "to", X_corr_removed.shape[1])

# Visualization of Correlation Matrix
plt.figure(figsize=(20,16))
correlation_matrix = X_encoded.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5, fmt=".2f", squa
plt.title('Feature Correlation Matrix')
plt.tight_layout();
```

```
3. Correlation-based Feature Removal:
Dropped features: ['Rating', 'Gender_Female', 'Student_Yes', 'Married_Yes']
Number of features reduced from 15 to 11
```




```
# 3. Correlation-based Feature Selection (with target)
def correlation_with_target_selection(X, y, threshold=0.2):
    # Calculate correlation with target
    correlations = X.apply(lambda col: col.corr(y))

    # Select features above the absolute correlation threshold
    selected_features = correlations[abs(correlations) >= threshold].index

    return X[selected_features], selected_features

# Apply Correlation with Target Selection
X_corr_target, corr_target_features = correlation_with_target_selection(X_encoded, y)
```

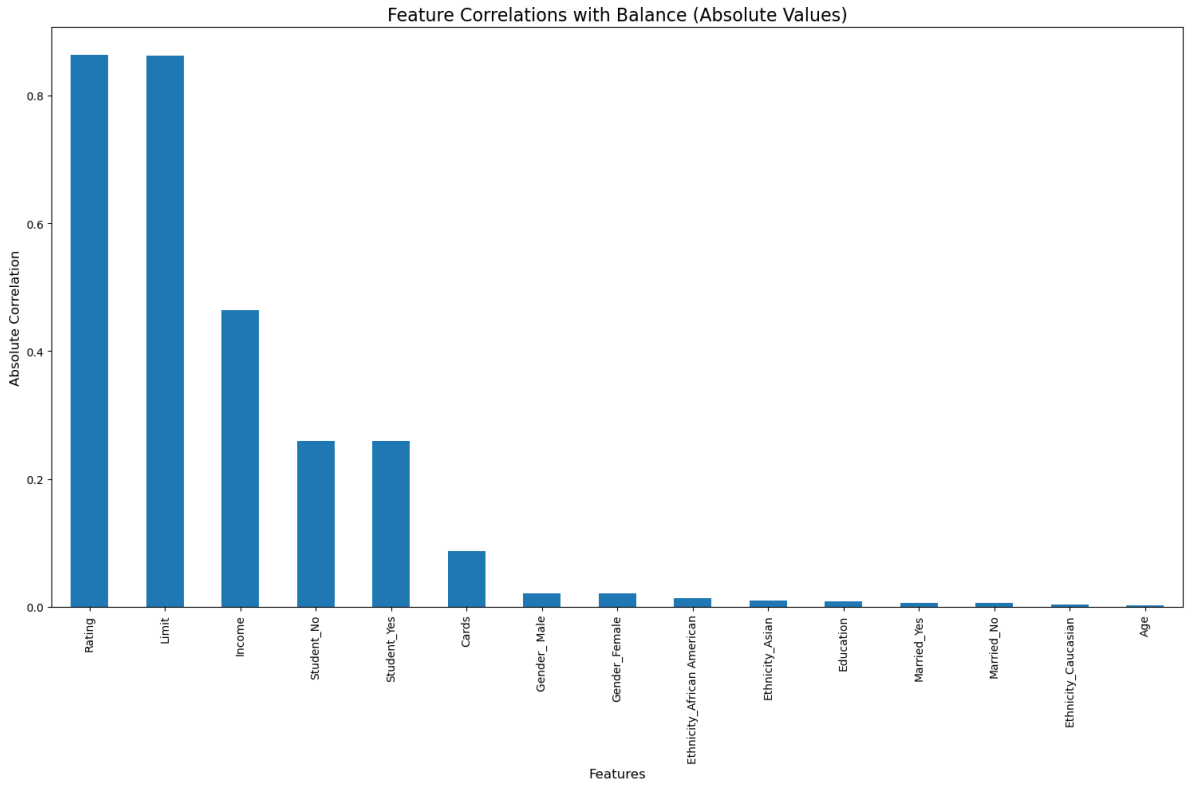
```
print("\n2. Correlation with Target Feature Selection:")
print("Selected features:", list(corr_target_features))
print("Number of features reduced from", X_encoded.shape[1], "to", len(corr_target_features))
```

```
2. Correlation with Target Feature Selection:
Selected features: ['Income', 'Limit', 'Rating', 'Student_No', 'Student_Yes']
Number of features reduced from 15 to 5
```

```
# Calculate correlations with target
correlations = X_encoded.apply(lambda col: col.corr(y))

# Sort correlations by absolute value in descending order
correlations_sorted = correlations.abs().sort_values(ascending=False)

# Prepare the visualization
plt.figure(figsize=(15, 10))
correlations_sorted.plot(kind='bar')
plt.title('Feature Correlations with Balance (Absolute Values)', fontsize=16)
plt.xlabel('Features', fontsize=12)
plt.ylabel('Absolute Correlation', fontsize=12)
plt.xticks(rotation=90)
plt.tight_layout()

# Print out the sorted correlations
print("Correlations with Balance (sorted by absolute value):")
for feature, corr in correlations_sorted.items():
    print(f"{feature}: {corr}")
```

```
Correlations with Balance (sorted by absolute value):
Rating: 0.863625160621495
Limit: 0.8616972670153951
Income: 0.46365645701575736
Student_No: 0.2590175474501476
Student_Yes: 0.2590175474501476
Cards: 0.08645634741861911
Gender_ Male: 0.021474006717338588
Gender_Female: 0.021474006717338588
Ethnicity_African American: 0.013719801718047214
Ethnicity_Asian: 0.00981223592782398
```

```
Education: 0.00806157645355343
Married_Yes: 0.005673490217239985
Married_No: 0.005673490217239973
Ethnicity_Caucasian: 0.003288321172097514
Age: 0.0018351188590736563
```




```python
# 4. SelectKBest Feature Selection
def select_k_best_features(X, y, k=5):
    # Create a SelectKBest selector
    selector = SelectKBest(score_func=f_regression, k=k)

    # Fit and transform the data
    X_selected = selector.fit_transform(X, y)

    # Get the selected feature names
    selected_features = X.columns[selector.get_support()]

    return X_selected, selected_features
```

```
# Apply SelectKBest
X_k_best, k_best_features = select_k_best_features(X_encoded, y)

print("\n4. SelectKBest Feature Selection:")
print("Selected features:", list(k_best_features))
print("Number of features reduced from", X_encoded.shape[1], "to", len(k_best_features))
```

```
4. SelectKBest Feature Selection:
Selected features: ['Income', 'Limit', 'Rating', 'Student_No', 'Student_Yes']
Number of features reduced from 15 to 5
```

12.3 Wrapper Methods

Wrapper methods evaluate subsets of features by actually training and testing a model on different feature combinations. These methods optimize feature selection based on model performance.

12.3.1 Characteristics:

- **Model-dependent:** They rely on training a model to assess feature usefulness.
- **Computationally expensive:** Since multiple models need to be trained, they can be slow, especially for large datasets.
- **More accurate than filter methods:** They account for feature interactions and can find the optimal subset.
- **Risk of overfitting:** Since they optimize for a specific dataset, the selected features may not generalize well to new data.
- **Common techniques:**
  - Recursive Feature Elimination (RFE)
  - Sequential Feature Selection (SFS) (Forward/Backward Selection)
- Wrapper methods can be explained with the help of following graphic:

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

12.3.2 Recursive Feature Elimination (RFE)

This is how RFE works:

- Start with all features in the dataset.
- Train a model (e.g., Random Forest, SVM, Logistic Regression).
- Rank feature importance using model coefficients (`coef_`) or feature importance scores (`feature_importances_`).
- Remove the least important feature(s).
- Repeat steps 2-4 recursively until reaching the desired number of features.
- Return the best subset of features.

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier


# Load dataset
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# Initialize a model
model = RandomForestClassifier(n_estimators=100, random_state=42)

# Apply Recursive Feature Elimination (RFE)
rfe_selector = RFE(model, n_features_to_select=5)  # Select top 5 features
X_rfe_selected = rfe_selector.fit_transform(X, y)

# Get selected feature names
rfe_selected_features = X.columns[rfe_selector.support_]
print("Selected features using RFE:", rfe_selected_features.tolist())
```

```
Selected features using RFE: ['mean concave points', 'worst radius', 'worst perimeter', 'wors
```

12.3.3 Recursive Feature Elimination with Cross-Validation (RFECV)

RFECV is an extension of Recursive Feature Elimination (RFE) that automatically selects the optimal number of features using cross-validation.

```python
from sklearn.feature_selection import RFECV
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold

# Load dataset
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# Initialize model (Random Forest)
model = RandomForestClassifier(n_estimators=100, random_state=42)

# Define cross-validation strategy
cv = StratifiedKFold(n_splits=5)

# Apply RFECV
rfecv_selector = RFECV(estimator=model, step=1, cv=cv, scoring='accuracy', n_jobs=-1)
X_rfecv_selected = rfecv_selector.fit_transform(X, y)

# Get selected feature names
rfecv_selected_features = X.columns[rfecv_selector.support_]
print("Optimal number of features:", rfecv_selector.n_features_)
print("Selected features using RFECV:", rfecv_selected_features.tolist())

# Print feature rankings
feature_ranking = pd.DataFrame({'Feature': X.columns, 'Ranking': rfecv_selector.ranking_})
print(feature_ranking.sort_values(by='Ranking'))
```

```
Optimal number of features: 16
Selected features using RFECV: ['mean radius', 'mean texture', 'mean perimeter', 'mean area'
                    Feature  Ranking
0               mean radius        1
22          worst perimeter        1
23               worst area        1
13               area error        1
24         worst smoothness        1
25        worst compactness        1
10             radius error        1
21            worst texture        1
26          worst concavity        1
7       mean concave points        1
6            mean concavity        1
```

```
3                 mean area        1
2            mean perimeter        1
1              mean texture        1
27     worst concave points        1
20             worst radius        1
28           worst symmetry        2
5          mean compactness        3
4           mean smoothness        4
29  worst fractal dimension        5
12          perimeter error        6
16          concavity error        7
15        compactness error        8
14         smoothness error        9
18           symmetry error       10
17     concave points error       11
19  fractal dimension error       12
11            texture error       13
8             mean symmetry       14
9    mean fractal dimension       15
```

12.3.4 Sequential Feature Selection

SFS iteratively selects the most relevant features by **adding them one at a time (forward selection)** or **removing them one at a time (backward elimination)** while evaluating model performance at each step.

The selection strategy is controlled using: - `direction='forward'` for **feature addition** - `direction='backward'` for **feature removal**

```
from sklearn.feature_selection import SequentialFeatureSelector

from sklearn.linear_model import LogisticRegression

# Initialize a model
model = LogisticRegression(max_iter=1000)

# Apply Sequential Feature Selection (SFS)
sfs_selector = SequentialFeatureSelector(model, n_features_to_select=5, direction='forward',
X_sfs_selected = sfs_selector.fit_transform(X, y)

# Get selected feature names
sfs_selected_features = X.columns[sfs_selector.get_support()]
print("Selected features using SFS:", sfs_selected_features.tolist())
```

```
Selected features using SFS: ['mean radius', 'radius error', 'worst texture', 'worst perimete
```

Let's use them on the *credit* dataset

```
# Load the data
df = pd.read_csv('./Datasets/Credit.csv', index_col=0)

# Separate features and target
X = df.drop('Balance', axis=1)
y = df['Balance']

# Preprocessing
# One-hot encode categorical variables
X_encoded = pd.get_dummies(X, columns=['Gender', 'Student', 'Married', 'Ethnicity'])

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X_encoded, y, test_size=0.2, random_state

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 1. Recursive Feature Elimination with Cross-Validation (RFECV)
def perform_rfecv(X_train, y_train, X_test, y_test):
    # Use both Linear Regression and Random Forest
    estimators = [
        ('Linear Regression', LinearRegression()),
        ('Random Forest', RandomForestRegressor(n_estimators=100, random_state=42))
    ]

    results = {}

    for name, estimator in estimators:
        # RFECV with cross-validation
        rfecv = RFECV(
            estimator=estimator,
            step=1,
            cv=5,
```

```
            scoring='neg_mean_squared_error'
        )

        # Fit RFECV
        rfecv.fit(X_train, y_train)

        # Get selected features
        selected_features = X_train.columns[rfecv.support_]

        # Prepare reduced datasets
        X_train_reduced = X_train[selected_features]
        X_test_reduced = X_test[selected_features]

        # Fit the model on reduced dataset
        estimator.fit(X_train_reduced, y_train)

        # Predict and evaluate
        y_pred = estimator.predict(X_test_reduced)
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)

        results[name] = {
            'Selected Features': list(selected_features),
            'Number of Features': len(selected_features),
            'MSE': mse,
            'R2': r2
        }

    return results

# 2. Sequential Feature Selection
def perform_sequential_feature_selection(X_train, y_train, X_test, y_test):
    # Use both Linear Regression and Random Forest
    estimators = [
        ('Linear Regression', LinearRegression()),
        ('Random Forest', RandomForestRegressor(n_estimators=100, random_state=42))
    ]

    results = {}

    for name, estimator in estimators:
        # Sequential Forward Selection
```

```
        sfs_forward = SequentialFeatureSelector(
            estimator=estimator,
            n_features_to_select='auto',  # will choose based on cross-validation
            direction='forward',
            cv=5,
            scoring='neg_mean_squared_error'
        )

        # Fit SFS
        sfs_forward.fit(X_train, y_train)

        # Get selected features
        selected_features = X_train.columns[sfs_forward.get_support()]

        # Prepare reduced datasets
        X_train_reduced = X_train[selected_features]
        X_test_reduced = X_test[selected_features]

        # Fit the model on reduced dataset
        estimator.fit(X_train_reduced, y_train)

        # Predict and evaluate
        y_pred = estimator.predict(X_test_reduced)
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)

        results[name] = {
            'Selected Features': list(selected_features),
            'Number of Features': len(selected_features),
            'MSE': mse,
            'R2': r2
        }

    return results

# Run feature selection methods
print("1. Recursive Feature Elimination with Cross-Validation (RFECV):")
rfecv_results = perform_rfecv(X_train, y_train, X_test, y_test)

print("\n2. Sequential Feature Selection:")
sfs_results = perform_sequential_feature_selection(X_train, y_train, X_test, y_test)
```

```python
# Print detailed results
def print_feature_selection_results(results):
    for model_name, result in results.items():
        print(f"\n{model_name}:")
        print(f"Number of Selected Features: {result['Number of Features']}")
        print("Selected Features:", result['Selected Features'])
        print(f"Mean Squared Error: {result['MSE']:.4f}")
        print(f"R2 Score: {result['R2']:.4f}")

print("\nRFECV Results:")
print_feature_selection_results(rfecv_results)

print("\nSequential Feature Selection Results:")
print_feature_selection_results(sfs_results)
```

```
1. Recursive Feature Elimination with Cross-Validation (RFECV):

2. Sequential Feature Selection:

RFECV Results:

Linear Regression:
Number of Selected Features: 15
Selected Features: ['Income', 'Limit', 'Rating', 'Cards', 'Age', 'Education', 'Gender_ Male'
Mean Squared Error: 7974.8564
R2 Score: 0.9523

Random Forest:
Number of Selected Features: 5
Selected Features: ['Income', 'Limit', 'Rating', 'Student_No', 'Student_Yes']
Mean Squared Error: 9124.1126
R2 Score: 0.9454

Sequential Feature Selection Results:

Linear Regression:
Number of Selected Features: 7
Selected Features: ['Income', 'Limit', 'Rating', 'Cards', 'Age', 'Student_No', 'Student_Yes']
Mean Squared Error: 8058.6421
R2 Score: 0.9518

Random Forest:
```

```
Number of Selected Features: 7
Selected Features: ['Income', 'Limit', 'Rating', 'Gender_Female', 'Student_No', 'Student_Yes
Mean Squared Error: 9953.4559
R2 Score: 0.9404
```

12.4 Embeded methods

Embedded methods integrate feature selection directly into the model training process. These methods learn which features are important while building the model, offering a balance between filter and wrapper methods.

12.4.1 Characteristics:

- **More efficient than wrapper methods:** Feature selection is built into the model training, avoiding the need for multiple iterations.
- **Less prone to overfitting than wrapper methods:** Regularization techniques help prevent overfitting.
- **Model-dependent:** The selected features are specific to the chosen model.
- **Common techniques:**
  - **Lasso (L1 Regularization):** Shrinks less important feature coefficients to zero.
  - **Decision Tree-based methods:** Feature importance scores from Random Forest, XGBoost, or Gradient Boosting.
  - **Elastic Net:** A combination of L1 and L2 regularization.

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

```python
from sklearn.linear_model import LassoCV
from sklearn.ensemble import RandomForestRegressor

# Load data
df = pd.read_csv('./Datasets/Credit.csv').drop(columns=df.columns[0])  # Drop index column

# Clean categorical variables (trim whitespace)
categorical_cols = ['Gender', 'Student', 'Married', 'Ethnicity']
for col in categorical_cols:
    df[col] = df[col].str.strip()

# Convert categorical variables to dummy variables
df = pd.get_dummies(df, columns=categorical_cols, drop_first=True)
```

```
# Separate features and target
X = df.drop(columns=['Balance'])
y = df['Balance']

# Split data into train/test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize features for Lasso
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Lasso Regression for feature selection
lasso = LassoCV(cv=5, random_state=42)
lasso.fit(X_train_scaled, y_train)

# Get non-zero Lasso coefficients
lasso_feat = pd.Series(lasso.coef_, index=X.columns)
selected_lasso = lasso_feat[lasso_feat != 0].sort_values(ascending=False)

# Random Forest for feature importance
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Get feature importances
rf_feat = pd.Series(rf.feature_importances_, index=X.columns)
selected_rf = rf_feat.sort_values(ascending=False)

# Display results
print("Lasso Selected Features (Non-Zero Coefficients):\n", selected_lasso)
print("\nRandom Forest Feature Importances:\n", selected_rf)
```

```
Lasso Selected Features (Non-Zero Coefficients):
 Limit                408.156450
Student_Yes           106.371166
Cards                  36.834149
Ethnicity_Caucasian     6.009746
Gender_Male            -0.254150
Age                   -28.253276
dtype: float64
```

```
Random Forest Feature Importances:
 Limit                  0.458433
Rating                 0.418695
Student_Yes            0.046555
Age                    0.022970
Unnamed: 0             0.018321
Education              0.014302
Cards                  0.010239
Married_Yes            0.003657
Gender_Male            0.002616
Ethnicity_Caucasian    0.002256
Ethnicity_Asian        0.001956
dtype: float64
```

12.5 Comparison of Feature Selection Methods

Method	Model Dependency	Computational Cost	Handles Feature Interactions	Risk of Overfitting
Filter	No	Low	No	Low
Wrapper	Yes	High	Yes	High
Embedded	Yes	Medium	Yes	Medium

12.6 Conclusion

- **Use Filter methods** when working with large datasets or when speed is a priority.
- **Use Wrapper methods** when accuracy is more important and computational cost is not a major concern.
- **Use Embedded methods** when leveraging models that inherently perform feature selection, such as Lasso regression or tree-based models.

By understanding these techniques, you can make better decisions when selecting features for machine learning models.

12.6.1 Reference:

- https://xavierbourretsicotte.github.io/subset_selection.html
- https://www.kaggle.com/code/prashant111/comprehensive-guide-on-feature-selection

- https://www.analyticsvidhya.com/blog/2016/12/introduction-to-feature-selection-methods-with-an-example-or-how-to-select-the-right-variables/

A Assignment 1

Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Do not write your name on the assignment.
3. Write your code in the **Code cells** and your answers in the **Markdown cells** of the Jupyter notebook. Ensure that the solution is written neatly enough to for the graders to understand and follow.
4. Use Quarto to render the **.ipynb** file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
5. There is a **bonus** question worth 15 points.
6. **Five points are properly formatting the assignment**. The breakdown is as follows:
    - Must be an HTML file rendered using Quarto **(1 point)**. *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file.*
    - No name can be written on the assignment, nor can there be any indicator of the student's identity—e.g. printouts of the working directory should not be included in the final submission. **(1 point)**
    - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) **(1 point)**
    - Final answers to each question are written in the Markdown cells. **(1 point)**
    - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text. **(1 point)**
7. The maximum possible score in the assigment is 100 + 15 (bonus question) + 5 (proper formatting) = 120 out of 100. There is no partial credit for some parts of the bonus question.

A.1 1) Case Studies: Regression vs Classification and Prediction vs Inference (16 points)

For each case below, explain (1) whether it is a classification or a regression problem and (2) whether the main purpose is prediction or inference. **You need justify your answers for credit.**

A.1.1 1a)

You work for a company that is interested in conducting a marketing campaign. The goal of your project is to identify individuals who are likely to respond positively to a marketing campaign, based on observations of demographic variables (such as age, gender, income etc.) measured on each individual. **(2+2 points)**

A.1.2 1b)

For the same company, now you are working on a different project. This one is focused on understanding the impact of advertisements in different media types on the company sales. For example, you are interested in the following question: *'How large of an increase in sales is associated with a given increase in radio and TV advertising?'* **(2+2 points)**

A.1.3 1c)

A company is selling furniture and they are interested in the finding the association between demographic characteristics of customers (such as age, gender, income etc.) and if they would purchase a particular company product. **(2+2 points)**

A.1.4 1d)

We are interested in forecasting the % change in the USD/Euro exchange rate using the weekly changes in the stock markets of a number of countries. We collect weekly data for all of 2023. For each week, we record the % change in the USD/Euro, the % change in the US market, the % change in the British market, and the % change in the German market. **(2+2 points)**

A.2 2) Examples for Different Regression Metrics: RMSE vs MAE (8 points)

A.2.1 2a)

Describe a regression problem, where it will be more proper to evaluate the model performance using the root mean squared error (RMSE) metric as compared to the mean absolute error (MAE) metric. **You need to justify your answer for credit. (4 points)**

Note: You are not allowed to use the datasets and examples covered in the lectures.

A.2.2 2b)

Describe a regression problem, where it will be more proper to evaluate the model performance using the mean absolute error (MAE) metric as compared to the root mean squared error (RMSE) metric. **You need to justify your answer for credit. (4 points)**

Note: You are not allowed to use the datasets and examples covered in the lectures.

A.3 3) Simple Linear Regression: Formulation (3 points)

When asked to state the simple linear regression model, a students wrote it as follows: $E(Y_i) = \beta_0 + \beta_1 X_i + \epsilon_i$. Is this correct **(1 point)**? Justify your answer **(2 points)**.

A.4 4) Modeling the Petrol Consumption in U.S. States (58 points)

Read **petrol_consumption_train.csv**. Assume that each observation is a U.S. state. For each observation, the data has the following variables as its five columns:

`Petrol_tax`: Petrol tax (cents per gallon)

`Per_capita_income`: Average income (dollars)

`Paved_highways`: Paved Highways (miles)

`Prop_license`: Proportion of population with driver's licenses

`Petrol_consumption`: Consumption of petrol (millions of gallons)

A.4.1 4a)

Create a pairwise plot of all the variables in the dataset. **(1 point)** Print the correlation matrix of all the variables as well. **(1 point)** Which variable has the highest linear correlation with `Petrol_consumption`? **(1 point)**

Note: Remember that a pairwise plot is a visualization tool that you can find in the seaborn library.

A.4.2 4b)

Fit a simple linear regression model to predict `Petrol_consumption` using the column you found in **part a** as the only predictor. Print the model summary. **(3 points)**

A.4.3 4c)

When asked for a point estimate of the expected petrol consumption for a state where the proportion of population with driver's license is 54.4%, a person gave the estimate 488 million gallons because that is the mean value of `Petrol_consumption` for the two observations of `Prop_license = 0.544` pieces in the dataset. Is there an issue with this approach? Explain. **(2 points)** If there is an issue, then suggest a better approach and use it to estimate the expected petrol consumption for a state where the proportion of population with driver's license is 54.4%. **(2 points)**

A.4.4 4d)

What is the increase in petrol consumption for an increase of 0.05 in the predictor? **(3 points)**

A.4.5 4e)

Does petrol consumption have a statistically significant relationship with the predictor? **You need to justify your answer for credit**. **(3 points)**

A.4.6 4f)

How much of the variation in petrol consumption can be explained by its linear relationship with the predictor? **(2 points)**

A.4.7 4g)

Predict the petrol consumption for a state in which 50% of the population has a driver's license. **(2 points)** What are the confidence interval **(2 points)** and the prediction interval **(2 points)** for your prediction? Which interval is wider? **(1 points)** Why? **(2 points)**

A.4.8 4h)

Predict the petrol consumption for a state in which 10% of the population has a driver's license. **(3 points)** Are you getting a reasonable outcome? **(1 point)** Why or why not? **(2 points)**

A.4.9 4i)

What is the residual standard error of the model? **(3 points)**

A.4.10 4j)

Using the trained model, predict the petrol consumption of the observations in **petrol_consumption_test.csv** **(2 points)** and find the RMSE. **(2 points)** What is the unit of this RMSE value? **(1 point)**

A.4.11 4k)

Based on the answers to **part i** and **part j**, do you think the model is overfitting? **You need to justify your answer for credit. (3 points)**

A.4.12 4l)

Make a scatterplot of `Petrol_consumption` vs. the predictor using **petrol_consumption_test.csv**. **(1 point)** Over the scatterplot, plot the regression line **(1 point)**, the prediction interval **(2 points)**, and the confidence interval. **(2 points)**

Make sure that regression line, prediction interval lines, and confidence interval lines have different colors. **(1 point)** Display a legend that correctly labels the lines as well. **(1 point)** Note that you need two lines of the same color to plot an interval.

A.4.13 4m)

The dataset consists of 40 US States. If you combine this data with the data of the remaining 10 US States, are you likely to obtain narrower confidence and prediction intervals in the plot developed in the previous question, for the same level of confidence? Justify your answer. **(2 points)**.

If yes, then can you gaurantee that the width of these intervals will reduce? Justify your answer. If no, then can you gaurantee that the width of these intervals will not reduce? Justify your answer. **(2 points)**

A.4.14 4n)

Find the correlation between `Petrol_consumption` and the rest of the variables in **petrol_consumption_train.csv**. Which column would have the lowest R-squared value when used as the predictor for a Simple Linear Regression model to predict `Petrol_consumption`? Note that you can directly answer this question from the correlation values and do not need to develop any more linear regression models. **(2 points)**

A.5 5) Reproducing the Results with Scikit-Learn (15 points)

A.5.1 5a)

Using the same datasets, same response and the same predictor as **Question 4**, reproduce the following outputs with scikit-learn:

- Model RMSE for test data **(3 points)**
- R-squared value of the model **(3 points)**
- Residual standard error of the model **(3 points)**

Note that you are only allowed to use scikit-learn, pandas, and numpy tools for this question. Any other libraries will not receive any credit.

A.5.2 5b)

Which of the model outputs from **Question 4** cannot be reproduced using scikit-learn? Give two answers. **(2+2 points)** What does this tell about scikit-learn? **(2 points)**

A.6 6) Bonus Question (15 points)

Please note that the bonus question requires you to look more into the usage of the tools we covered in class and it will be necessary to do your own research. We strongly suggest attempting it after you are done with the rest of the assignment.

A.6.1 6a)

Fit a simple linear regression model to predict `Petrol_consumption` based on the predictor in **Question 4**, but **without an intercept term**. **(5 points - no partial credit)**

Without an intercept means that the equation becomes $Y = \beta_1 X$. The intercept term, β_0, becomes 0.

Note: You must answer this part correctly to qualify for the bonus points in the following parts.

A.6.2 6b)

Predict the petrol consumption for the observations in **petrol_consumption_test.csv** using the model without an intercept and find the RMSE. **(1+2 points)** Then, print the summary and find the R-squared. **(2 points)**

A.6.3 6c)

The RMSE for the models with and without the intercept are similar, which indicates that both models are almost equally good. However, the R-squared for the model without intercept is much higher than the R-squared for the model with the intercept. Why? Justify your answer. **(5 points - no partial credit)**

B Assignment 2

Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the **Code cells** and your answers in the **Markdown cells** of the Jupyter notebook. Ensure that the solution is written neatly enough to for the graders to understand and follow.
3. Use Quarto to render the **.ipynb** file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. **Five points are properly formatting the assignment**. The breakdown is as follows:
    - Must be an HTML file rendered using Quarto **(1 point)**. *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file.*
    - No name can be written on the assignment, nor can there be any indicator of the student's identity—e.g. printouts of the working directory should not be included in the final submission. **(1 point)**
    - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) **(1 point)**
    - Final answers to each question are written in the Markdown cells. **(1 point)**
    - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text. **(1 point)**
5. The maximum possible score in the assigment is 100 + 5 (proper formatting) = 105 out of 100.

B.1 1) Multiple Linear Regression (24 points)

A study was conducted on 97 male patients with prostate cancer who were due to receive a radical prostatectomy (complete removal of the prostate). The **prostate.csv** file contains data on 9 measurements taken from these 97 patients. Each row (observation) represents a patient and each column (variable) represents a measurement. The description of variables can be found here: https://rafalab.github.io/pages/649/prostate.html

B.1.1 1a)

Fit a linear regression model with `lpsa` as the response and all the other variables as the predictors. Print its summary. **(2 points)** Write down the **optimal equation** that predicts `lpsa` using the predictors. **(2 points)**

B.1.2 1b)

Is the **overall regression** statistically significant? In other words, is there a statistically significant relationship between the response and at least one predictor? **You need to justify your answer for credit. (2 points)**

B.1.3 1c)

What does the optimal coefficient of `svi` tell us as a numeric output? Make sure you include the predictor, (`svi`) the response (`lpsa`) and the other predictors in your answer. **(2 points)**

B.1.4 1d)

Check the p-values of `gleason` and `age`. Are these predictors statistically significant? **You need to justify your answer for credit. (2 points)**

B.1.5 1e)

Check the 95% Confidence Interval of `age`. How can you relate it to its p-value and statistical significance, which you found in the previous part? **(2 points)**

B.1.6 1f)

This question requires some thinking, and bringing your 303-1 and 303-2 knowledge together.

Fit a **simple** linear regression model on `lpsa` against `gleason` and check the p-value of `gleason` using the summary. **(2 point)** Did the statistical significance of `gleason` change in the absence of other predictors? **(1 point)** Why or why not? **(3 points)**

Hints:

1) You need to compare this model with the Multiple Linear Regression model you created above.
2) Printing a correlation matrix of all the predictors should be useful.

B.1.7 1g)

Predict the `lpsa` of a **65 year old** man with `lcavol` = 1.35, `lweight` = 3.65, `lbph` = 0.1, `svi` = 0.22, `lcp` = -0.18, `gleason` = 6.75, and `pgg45` = 25. Find the 95% confidence and prediction intervals as well. **(2 points)**

B.1.8 1h)

In the Multiple Linear Regression model with all the predictors, you should see a total of five predictors that appear to be statistically insignificant. Why is it not a good idea to directly conclude that all of them are statistically insignificant? **(2 points)** Implement the additional test that concludes the statistical insignificance of all five predictors. **(2 points)**

Hint: `f_test()` method

B.2 2) Multiple Linear Regression with Variable Transformations (22 points)

The **infmort.csv** file has the infant mortality data of different countries in the world. The `mortality` column represents the infant mortality rate with "deaths per 1000 infants" as the unit. The `income` column represents the per capita income in USD. The other columns should be self-explanatory. (This is an old dataset, as can be seen from some country names.)

B.2.1 2a)

Start your analysis by creating (i) a boxplot of log(`mortality`) for each `region` and (ii) a boxplot of `income` for each `region`. Note that the `region` column has the continent names. **(3 points)**

Note: You need to use np.log, which is the natural log. This is to better distinguish the mortality values.

B.2.2 2b)

In the previous part, you should see that Europe has the lowest infant mortality rate on average, but it also has the highest per capita income on average. Our goal is to see if Europe still has the lowest mortality rate if we remove the effect of income. We will try to find an answer for the rest of this question.

Create four scatter plots: (i) `mortality` against `income`, (ii) log(`mortality`) against `income`, (iii) `mortality` against log(`income`), and (iv) log(`mortality`) against log(`income`). **(3 points)** Based on the plots, create an appropriate model to predict the mortality rate as a function of per capita income. Print the model summary. **(2 points)**

B.2.3 2c)

Update the model you created in the previous part by adding `region` as a predictor. Print the model summary. **(2 points)**

B.2.4 2d)

Use the model developed in the previous part to compute a new `adjusted_mortality` variable for each observation in the data. **(5 points)** Adjusted mortality rate is the mortality rate after removing the estimated effect of income. You need to calculate it with the following steps:

- Multiply the (transformed) income column with its optimal coefficient. This is the estimated effect of income.
- Subtract the product from the (transformed) response column. This removes the estimated effect of income.
- You may need to do a inverse transformation to calculate the actual adjusted mortality rate values.

Make a boxplot of log(`adjusted_mortality`) for each `region`. **(2 points)**

B.2.5 2e)

Using the plots in parts **a** and **d**, answer the following questions:

(i) Does Europe still have the lowest mortality rate on average after removing the effect of income?

(ii) How did the distribution of values among different continents change after removing the effect of income? How did the comparison of different continents change? Does any non-European country have a lower mortality rate than all the European countries after removing the effect of income?

(5 points)

B.3 3) Variable Transformations and Interactions (38 points)

The **soc_ind.csv** dataset contains many social indicators of a number of countries. Each row is a country and each column is a social indicator. The column names should be clear on what the variables represent. The GDP per capita will be the response variable throughout this question.

B.3.1 3a)

Using correlations, find out the most useful predictor for a simple linear regression model with `gdpPerCapita` as the response. You can ignore categorical variables for now. Let that predictor be P. **(2 points)**

B.3.2 3b)

Create a scatterplot of `gdpPerCapita` vs P. Does the relationship between `gdpPerCapita` and P seem linear or non-linear? **(2 points)**

B.3.3 3c)

If the relationship in the previous part is non-linear, create three models:

- Only with P
- With P and its quadratic term
- With P, its quadratic term and its cubic term

(2x3 = 6 points)

Compare the R-squared values of the models. **(2 points)**

B.3.4 3d)

On the same figure:

- create the scatterplot in part b.
- plot the linear regression line (only using P)
- plot the polynomial regression curve that includes the quadratic and cubic terms.
- add a legend to distinguish the linear and cubic fits.

(6 points)

B.3.5 3e)

Develop a model to predict `gdpPerCapita` using P and `continent` as predictors. (No higher-order terms.)

1. Which continent creates the baseline? **(2 points)** Write down its equation. **(2 points)**
2. For a given value of P, are there any continents that **do not** have a statistically significant difference of predicted `gdpPerCapita` from the baseline continent? If yes, then which ones, and why? If no, then why not? You need to justify your answers for credit. **(4 points)**

B.3.6 3f)

The model developed in the previous part has a limitation. It assumes that the increase in predicted `gdpPerCapita` with a unit increase in P does not depend on the `continent`.

Eliminate this limitation by including the interaction of `continent` with P in the model. Print the model summary of the model with interactions. **(2 points)** Which continent has the closest increase in predicted `gdpPerCapita` to the baseline continent with a unit increase in P. Which continent has the furthest? **You need to justify your answers for credit. (5 points)**

B.3.7 3g)

Using the model developed in the previous part, plot the regression lines of all the continents on the same figure. Put `gdpPerCapita` on the y-axis and P on the x-axis. **(4 points)** Use a legend to color-code the continents. **(1 point)**

B.4 4) Prediction with Sklearn (20 points)

B.4.1 Instructions

1. Read the **soc_ind.csv** dataset and use the `Index` column as the index of the dataframe.
2. Drop the `geographic_location` and `country` columns since they are not useful for our prediction.
3. We will use **only `sklearn` and `pandas`** libraries in this task.
4. **gdpPerCapita** is the response (target) variable.
   (2 points)

5. All other columns (except `Index`, `geographic_location`, and `country`) will serve as predictors.
   (2 points)

6. The `continent` column must be one-hot-encoded using `OneHotEncoder`.
   (5 points)
7. Output the encoded dummy variables (feature names) to confirm successful encoding.
   (2 points)

8. Train a `LinearRegression` model. Split the dataset into a **training set (90%)** and a **test set (10%)**.
   - Set `random_state=3` for reproducibility.
     (3 points)

9. Calculate the **RMSE** and **R-squared** for both the training set and the test set.
   (3 points)

10. Discuss whether your model shows signs of **overfitting** or **underfitting**.
    (3 points)

C Assignment 3

Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the *Code* cells and your answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
3. Use Quarto to print the *.ipynb* file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. **Five points are properly formatting the assignment**. The breakdown is as follows:
    - Must be an HTML file rendered using Quarto **(1 point)**. *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file.*
    - No name can be written on the assignment, nor can there be any indicator of the student's identity—e.g. printouts of the working directory should not be included in the final submission. **(1 point)**
    - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) **(1 point)**
    - Final answers to each question are written in the Markdown cells. **(1 point)**
    - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text. **(1 point)**

Data description

The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls, where bank clients were called to subscribe for a term deposit.

There is one train data - *train.csv*, which you will use to develop a model. There are two test datasets - *test1.csv* and *test2.csv*, which you will use to test your model. Each observation is a phone call and each column is a variable about the client or the phone call. Each dataset has the following attributes about the clients called in the marketing campaign:

1. `age`: Age of the client
2. `education`: Education level of the client
3. `day`: Day of the month the call is made
4. `month`: Month of the call
5. `y`: did the client subscribe to a term deposit?
6. `duration`: Call duration, in seconds. This attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the `duration` is not known before a call is performed. Also, after the end of the call `y` is obviously known. Thus, this input should only be included for inference purposes and should be discarded if the intention is to have a realistic predictive model.

(Source: UCI Data Archive. Please use the given datasets for the assignment, not the raw data from the source. It is just for reference.)

Instructions / suggestions for answering questions

(1) **Instruction:** Use *train.csv* for all questions, unless otherwise stated.

(2) **Suggestion 1:** You may use the functions in the class notes for printing the confusion matrix and the overall classification accuracy based on test / train data.

(3) **Suggestion 2:**: If you make variable transformations, you will need to do it for all the three datasets. Your code will be a bit concise if you make a function containing all the transformations, and then call it for the training and the two test datasets. You can put this function in the beginning of the code and keep adding transformations to it as you proceed with the assignment. You may need transformations in questions (1) and (13).

C.1 1)

Read the datasets. Make an appropriate visualization to visualize how the proportion of clients subscribing to a term deposit change with increasing call duration.

(4 points)

Hints:

1. Bin `duration` to create `duration_binned`. Group the data to find the fraction of clients responding positively to the marketing campaign for each bin in `duration_binned`. Make a lineplot of percentage of clients subscribing to a term deposit vs `duration_binned`, where the bins in `duration_binned` are arranged in increasing order of duration.
2. You may choose an appropriate number of bins & type of binning that helps you visualize well.
3. You may also think of other ways of visualization. You don't need to stick with this one.

C.2 2) Predictor `duration`

Based on the plot in (1), comment whether `duration` seems to be a useful variable to predict if the client will subscribe to a term deposit.

(1 point)

C.3 3) Model based on `duration`

Develop a logistic regression model to predict if the client subscribed to a term deposit based on call `duration`. Use the model to make a lineplot showing the probability of the client subscribing to a term deposit based on call `duration`.

(3 points)

Note

Answer questions 4 to 11 based on the regression model developed in (3).

C.4 4) Model significance

Is the regression model in statistically significant? Justify your answer.

(1 point for code, 1 point for answer)

C.5 5) Subscription probability in 5 minutes

What is the probability that the client subscribes to a term deposit with a 5-minute marketing call? Note that the call `duration` in data is given in *seconds*.

(2 points)

C.6 6) Call duration for subscription

What is the minimum call duration (in minutes) for which a client has a 95% or higher chance of subscribing to a term deposit?

(3 points)

C.7 7) Maximum call duration

What is the maximum call duration (in minutes) in which a client refused to subscribe to a term deposit? What was the probability of the client subscribing to the term deposit in that call?

(3 points)

C.8 8) Percent increase in odds

What is the percentage increase in the odds of a client subscribing to a term deposit when the call `duration` increases by a minute?

(3 points)

C.9 9) Doubling the subscription odds

How much must the call `duration` increase (in minutes) so that it doubles the odds of the client subscribing to a term deposit.

(3 points)

C.10 10) Classification accuracy

What is minimum overall classification accuracy of the model among the classification accuracies on *train.csv*, *test1.csv* and *test2.csv*? Consider a threshold of 30% when classifying observations.

(2 + 1 + 1 points)

C.11 11) Recall

What is the minimum *Recall* of the model among the *Recall* performance on *train.csv*, *test1.csv* and *test2.csv*? Consider a decision threshold probability of 30% when classifying observations.

Here, *Recall* is the proportion of clients predicted to subscribe to a term deposit among those who actually subscribed.

(3 points)

C.12 12) Subscription probability based on `age` and `education`

Develop a logistic regression model to predict the probability of a client subscribing to a term deposit based on `age`, `education` and the two-factor interaction between `age` and `education`. Based on the model, answer:

a. People with which type of `education` *(primary / secondary / tertiary / unknown)* have the highest percentage increase in odds of subscribing to a term deposit with a unit increase in `age`? Justify your answer.

b. What is the percentage increase in odds of a person subscribing to a term deposit for a unit increase in `age`, if the person has *tertiary* `education`.

c. What is the percentage increase in odds of a person subscribing to a term deposit for a unit increase in `age`, if the person has *primary* `education`.

(1 point for developing the model, 3 points for (a), 3 points for (b), 3 points for (c))

C.13 13) Model development

Develop a logistic regression model *(using train.csv)* to predict the probability of a client subscribing to a term deposit based on `age`, `education`, `day` and `month`. The model must have:

a. Minimum overall classification accuracy of 75% among the classification accuracies on *train.csv*, *test1.csv* and *test2.csv*.

b. Minimum recall of 50% among the recall performance on *train.csv*, *test1.csv* and *test2.csv*.

For all the three datasets - *train.csv*, *test1.csv* and *test2.csv*, print the:

1. Model summary (only for *train.csv*),
2. Confusion matrices,
3. Overall classification accuracies, and
4. Recall

Note that:

1. You cannot use `duration` as a predictor because its value is determined after the marketing call ends. However, after the call ends, we already know whether the client responded positively or negatively. That is why we have used `duration` only for inference in the previous questions. It helped us understand the effect of the length of the call on marketing success.
2. It is possible to develop the model satisfying constrains (a) and (b) with just appropriate transformation(s) of the predictor(s). However, you may consider interactions if you wish. Justify the transformations, if any, with visualizations.
3. You are free to choose any value of the decision threshold probability for classifying observations. However, you must use the same threshold on all the three datasets.

(10 points)

C.14 14) ROC-AUC

Report the probability that the model will predict a higher probability of response for a customer who signs up for the term deposit as compared to the customer who does not sign up, i.e., the ROC-AUC of the developed model in (13).

Hint: Use the functions `roc_curve`, and `auc` from the `sklearn.metrics` module

(3 points)

C.15 15) Net-profit

Suppose that the model developed in (13) is used to predict the clients in *test1.csv* and *test2.csv* who will respond positively to the campaign. Only those clients who are predicted to respond positively are called during the marketing campaign. Assume that:

1. A profit of \$100 is associated with a client who responds positively to the campaign,
2. A loss of \$10 is associated with a client who responds negatively to the campaign

What is the net profit from the campaign? Use the confusion matrices printed in (13).

(4 points)

C.16 16) Decision threshold probability

Based on the profit and loss associated with client responses specified in (15), and the model developed in (13), find the decision threshold probability of classification, such that the net profit is maximized. Use *train.csv*

Proceed as follows:

1. You would have obtained FPR and TPR for all potential decision threshold probabilities in (14).
2. Formulate an expression quantifying the net profit per client, in terms of FPR, TPR, and the overall response rate, i.e., proportion of people actually subscribing to the term deposit.
3. Find the decision threshold probability that maximizes the expression in (2).

(5 points)

C.17 17) Net profit based on new decision threshold probability

Using the new decision threshold probability obtained in (16), answer (15), i.e., what is the net-profit associated with the clients in *test1.csv* and *test2.csv* if a marketing campaign is performed. Again, only those clients who are predicted to respond positively, based on the new decision threshold probability, are called during the marketing campaign

Also, print the confusion matrices for predictions on *test1.csv* and *test2.csv* with the new threshold probability.

(4 points)

C.18 18) Model preference

Was the classification accuracy of the model in (13) higher than that of the model in (17)? If yes, then should you prefer the model in (13) for the marketing campaign? Why or why not?

Note: The model in (17) is the same as in (13), except with a different decision threshold probability

(3 points)

C.19 19) ROC curve

Plot the ROC curve for the model developed in (13). Mark the point on the curve corresponding to the decision threshold probability identified in (16).

Note that the ROC curve is independent of the decision threshold probability used by the model for prediction

(3 points)

C.20 20) Profit with TPR / FPR

Make a scatterplot of TPR vs FPR, and color the points based on net profit per client.

You can use the following code to make the plot if you have the relevant metrics in `tpr`, `fpr`, and `net_profit`

(1 point)

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.5)
plt.rcParams["figure.figsize"] = (9,6)
plt.rcParams["figure.autolayout"] = True
f, ax = plt.subplots()
points = ax.scatter(fpr, tpr, c = net_profit, s=50, cmap="Blues")
f.colorbar(points, label = "Net profit ($) \n(per client)")
plt.xlabel("False positive rate")
plt.ylabel("True positive rate")
plt.show()
```

C.21 21) Precision-recall

Compare the precision and recall of the models in (13) and (17) on *train.csv*.

Note: The model in (17) is the same as in (13), except with a different decision threshold probability

(4 points)

C.22 22) Precision-recall: important metric

Based on the above comparison, which metric among precision and recall turns out to be more important for maximizing the net profit in the marketing campaign?

(1 point)

C.23 23) Precision-recall curve

Plot the precision-recall curve vs decision threshold probability for the model developed in (13). Mark the points on the curve corresponding to the decision threshold probability identified in (16).

(3 points)

C.24 24) Precision-recall vs FPR-TPR

Instead of using the FPR and TPR metrics to find the optimum decision threshold probability in (16), use the precision-recall metrics to find the same.

(5 points)

C.25 25) Sklearn

Using train.csv and **only sklearn, pandas, and numpy**, train a Logistic Regression model. You need the following steps:

- The response is still y.
- Predictors are education, month, day and age.

- Numerical predictors need to be transformed to **all** their second-order polynomial versions.

- Logistic regression benefits from feature scaling, especially when using polynomial features, as it helps the optimization algorithm converge faster. Use StandardScaler or MinMaxScaler to scale the features.
- Categorical predictors need to be one-hot-encoded. They should not interact with the numerical predictors.

Print the accuracy and recall for both training and test data using a threshold of 0.11. Use **test1.csv** as the test dataset. **Remember that the test dataset needs to go through the exact same transformation as the training dataset.**

Hints:

- Do not scale categorical features: One-hot encoded dummy variables are already on the same scale (0 or 1). Scaling them is unnecessary and may distort their meaning.
- Scale numerical features: This includes both original numerical features and any polynomial features generated from them.
- Use a structured approach: A production-ready way to handle these transformations is by using ColumnTransformer within a Pipeline, ensuring that preprocessing steps are applied consistently. However, using a pipeline is optional.

(8 points)

D Assignment 4

Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the **Code cells** and your answers in the **Markdown cells** of the Jupyter notebook. Ensure that the solution is written neatly enough to for the graders to understand and follow.
3. Use Quarto to render the **.ipynb** file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. There is a **bonus** question worth 11 points.
5. **Five points are properly formatting the assignment**. The breakdown is as follows:
    - Must be an HTML file rendered using Quarto **(1 point)**. *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file.*
    - No name can be written on the assignment, nor can there be any indicator of the student's identity—e.g. printouts of the working directory should not be included in the final submission. **(1 point)**
    - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) **(1 point)**
    - Final answers to each question are written in the Markdown cells. **(1 point)**
    - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text. **(1 point)**
6. The maximum possible score in the assigment is 99 + 11 + 5 = 115 out of 100.

D.1 1) Modeling the Radii of Exoplanets (40 points)

For this question, we are interested in predicting the radius of exoplanets (planets outside the Solar System) in kilometers. To achieve this goal, we will use NASA's Composite Planetary Systems dataset and different regression models. (See https://exoplanetarchive.ipac.caltech.edu for more context.)

Read all three CompositePlanetarySystems datasets - you should have one training and two test datasets. Each row is an exoplanet. `pl_rade` column represents the radius of each exoplanet as a proportion of Earth's radius, which is approximately 6,378 km.

D.1.1 a)

Develop a linear regression model (no non-linear terms) to predict `pl_rade` using all the raw variables in the data except `pl_name`, `disc_facility` and `disc_locale`. You can use either `statsmodels` or `sklearn`. **(2 points)**

D.1.2 b)

Find the `RMSE` of the model using both test sets **separately**. (You need to print two `RMSE` values.) Note that the library you used should not make a difference here! **(2 points)**

Print the training `RMSE` as well for reference. **(1 point)**

D.1.3 c)

Compare the training and test `RMSEs`. **(1 point)** What is the issue with this model? **(1 point)**

D.1.4 d)

Train a **Ridge regression model** to predict `pl_rade` using the same set of variables as above. Optimize the regularization strength (`alpha`) using `RidgeCV` with **5-fold cross-validation**, ensuring that the data is **shuffled** before splitting (`random_state=42`). Use `neg_root_mean_squared_error` as the scoring metric.

Note:

- Scaling is essential for regularization to ensure fair weighting of features.
- Use the following range of **alpha** for hyperparamter tuning: `alphas = np.logspace(2,0.5,200)`

D.1.5 e)

Using the optimized model, print the `RMSEs` for the training set and both test sets. **(4 points)**

D.1.6 f)

How did the training and test performance change? Explain why the Ridge regression changed the training and test results. **(3 points)**

D.1.7 g)

Find the predictor whose coefficient is shrunk by far the most by Ridge regularization. **(3 points)**

Hint: `.coef_` and `.columns` attributes should be helpful.

D.1.8 h)

Why did the coefficient of the predictor identified in the previous question shrunk the most? Justify your answer for credit. **(2 points)**

Hint: Correlation vector/matrix

D.1.9 i)

Visualize how the coefficients change with the change in the hyperparameter value:

- Create a line plot of coefficient values vs. the hyperparameter value.
- Color code each predictor's coefficient values.
- Use log scale where necessary.
- Use an alphas vector of `np.logspace(7,0,200)` for better visualization

(5 points)

D.1.10 j)

Replace the Ridge regression with Lasso regression.

- Find the optimal hyperparameter using `LassoCV` **(2 points)**.
  - You need a different hyperparameter array - use: `np.logspace(0,-2.5,200)`
  - Using the same splitting strategy as Ridge regression
  - Note: The `Lasso` object does not have a `scoring` hyperparameter.
- Using the optimized Lasso model, print the `RMSEs` for the training set and both test sets. **(2 points)**
- Visualize how the Lasso coefficients change with alpha. **(2 points)**
  - You may use the range of alpha values as `np.logspace(7,-2.5,200)` for better visualization.

D.1.11 k)

Using the two figures created in parts i and j, explain how the Ridge and Lasso models behave differently as the hyperparameter value changes. **(2 points)** What does that difference mean for the usage of the Lasso model? **(1 point)**

D.1.12 l)

Find the predictors that are eliminated by Lasso regularization. **(2 points)**

D.2 2) Enhancing House Price Prediction with Higher-Order Terms and Cross-Validation (29 points)

In this question, we are interested in improving the prediction performance for house prices using five predictors.

D.2.1 a)

Read the house feature and price files and create the training and test datasets. The response is log-price and the five predictors are the rest of the variables, except house_id. **(2 points)**

D.2.2 b)

Previously, we observed that a linear model using raw predictors fails to capture the complexity of the problem, resulting in **underfitting**. Our goal is to examine how training and test performance evolve as model complexity increases.

Task Breakdown:

1. **Generate Higher-Order Features**
   - Use `PolynomialFeatures` from `sklearn` to create higher-order versions of the predictors (including both transformations and interactions) for both the training and test datasets. **(3 points)**
2. **Train a Ridge Regression Model**
   - Use all predictors (original and transformed) to train a **Ridge regression model** with `alpha = 0.000001`. **(2 points)**
3. **Compute RMSE Scores**
   - Store the **RMSE** for both the training and test sets. **(2 points)**
4. **Repeat for Different Polynomial Orders**
   - Perform this process for polynomial orders ranging from **1 to 6**. **(2 points)**
5. **Visualize the Performance**
   - Plot the **training and test RMSE values** as a function of polynomial order. Ensure:
     - The two curves have distinct colors. **(1 points)**
     - A legend is included for clarity. **(1 points)**

Notes:

- **Exclude the bias term** that `PolynomialFeatures` adds by default.
- **Feature scaling is required** for regularization. **(2 points)**
- **Minimal regularization is necessary** to prevent the test RMSE from diverging to infinity at higher polynomial orders, unlike in pure linear regression.

D.2.3 c)

Which order has the best test RMSE? **(1 point)** What is the best test RMSE? **(1 point)** At which order does the overfitting start? **(1 point)**

D.2.4 d)

Repeat part b, only this time use `RidgeCV` to find the best amount of regularization for each order by cross-validation. Use `alphas = np.logspace(2,0.5,200)` and `LOOCV`. Use `neg_root_mean_squared_error` for scoring. Create the same plot as part b. **(4 points)** Describe the obvious difference between the plot in this part and the plot in part b. **(2 points)**

D.2.5 e)

What is the best test RMSE found by using higher-orders and regularization? **(1 point)** Which order achieved this test RMSE? **(1 point)** Why did this order with regularization perform better than any lower order with or (almost) without regularization? **(3 points)**

D.3 3) Systematic Elimination of Interaction Terms (30 points)

In this question, we are interested in predicting if the client subscribed to a term deposit or not after a phone call using **age** and **education** of the client and the **day** and the **month** the call took place.

Note that this is the same problem as in the previous assignment, however, **using sklearn**, we aim to make the predictive analysis with interactions more systematic.

D.3.1 a)

Read `train.csv`, `test1.csv`, and `test2.csv`. Prepare the training and two test datasets according the description above. **(2 points)**

D.3.2 b)

For all datasets:

- One-hot-encode the categorical predictors. **(2 points)**
- Get the interactions of **all** the predictors. (Numeric and one-hot-encoded) **(3 points)**
  - Note that there is a very quick way of doing this with `PolynomialFeatures`
  - Don't forget to exclude the bias.
- Scale the predictors (correctly.) **(2 points)**

D.3.3 c)

Train a Logistic Regression model with Lasso penalty. **(2 points)** The idea is to discard interactions that are not useful. Note that instead of the manual, trial-and-error way of adding interactions in statsmodels, we include all the possible interactions and then discard the useless ones here.

- Use `[0.0001,0.001, 0.01, 0.1, 1, 10, 100, 1000]` as the possible `C` values. **(1 point)**
- Use 10-fold cross-validation to optimize the `C` value. **(1 point)**
- Lasso is very useful, but it needs special algorithms, since it includes non-differentiable absolute values. Use `saga` as the solver. **(1 point)**
- The default number of iterations the algorithm takes is usually not enough for `Lasso`. Use `max_iter = 1000`. (Default is 500) **(1 point)**
- **This will take 10-20 minutes to run.**

D.3.4 d)

How many models in total are run by this cross-validation process? **(2 points)**

D.3.5 e)

What is the optimum `C` value? **(1 point)** What is the `lambda` (in the Lasso cost) value it corresponds to? **(1 point)**

D.3.6 f)

What is the percentage of terms (linear or interaction) that are discarded by Lasso? (**Hint:** `.coef_`) **(2 points)**

D.3.7 g)

Find the five terms that have the highest effect on the logodds of a subscription. Assume that we are quantifying the effect of a term with the **absolute** value of its coefficient. (**Hint:** `.get_feature_names_out()`) **(4 points)**

D.3.8 h)

Come up with real-life explanations on why the terms identified in the previous part are important. (This is an open-ended question, just make sure your answer makes sense.) **(2 points)**

D.3.9 i)

Lastly, tune the classification threshold to get both test datasets above 75% accuracy and 50% recall. Note that you only worry about the threshold now. Lasso took care of finding good interactions. **(3 points)**

D.4 4) Bonus: ElasticNet (11 points)

The goal of this section is to familiarize you with `ElasticNet`, which combines both `Ridge` and `Lasso` regularization techniques. The balance between the Lasso and Ridge penalties is controlled by a hyperparameter.

D.4.1 a)

For regression, scikit-learn provides both `ElasticNet` and its cross-validation counterpart, `ElasticNetCV`, for implementing Elastic Net regularization.

Your tasks:

- Use the same dataset as in Question 1, ensuring that the specified columns remain **dropped**.
- Use the same performance metric: `RMSE`
- Apply the **same data splitting strategy** as in **Question 1**.
- Train an **Elastic Net model** with the following `l1_ratio` values:
  - **25% Lasso / 75% Ridge** (`l1_ratio = 0.25`)
  - **50% Lasso / 50% Ridge** (`l1_ratio = 0.50`)

  - **75% Lasso / 25% Ridge** (`l1_ratio = 0.75`)

- Tune the **alpha** hyperparameter using the range:

```
alphas = np.logspace(10, 0.1, 200)
```

- Combine the two test sets.
- Identify the `l1_ratio` and `alpha` combination that yields the best test performance.

(8 points)

D.4.2 b)

How many models were run in the cross-validation process of two hyperparameters? **(1 point)**

D.4.3 c)

Briefly describe how you would implement `ElasticNet` for Logistic Regression in scikit-learn. **(2 points)**

E Datasets, assignment and project files

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